The Report Committee for Sadaf Hussain Syed
Certifies that this is the approved version of the following report:

Smart Closet Inventory: NFC based inventory management solution

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Adnan Aziz

Christine Julien

Smart Closet Inventory: NFC based inventory management solution

by

Sadaf Hussain Syed, B.S.C.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2015

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Adnan Aziz for his encouragement, motivation, and technical guidance on this project. I could not have imagined having a better advisor. A special acknowledgment goes to Dr. Christine Julien for being a reader for this report, and an inspiration in many ways. I would also like to thank Amy Reed for her efforts and technical contributions to this project. I am deeply grateful to my supervisor at Cisco, Regis Marco, whose continuous support was instrumental to the successful completion of my Master's coursework. I am also indebted to my friend, Caleb Mayeux, for always giving me words of encouragement and for carefully proofreading my report and valuable suggestions and corrections. Finally, I would like to extend special thanks to my parents, Hussain Syed and Nisar Fatima, and my sister, Saba Syed, for their confidence, encouragement, and unconditional support throughout my study.

Abstract

Smart Closet Inventory: NFC based usage tracking system to organize clothes

Sadaf Hussain Syed, MSE

The University of Texas at Austin, 2015

Supervisor: Adnan Aziz

Clothes become worn, their colors fade, and sometimes they do not fit well. Wardrobes and closets require reorganization every season to remove unused articles. Smart Closet Inventory (SCI) is an easy to use system allowing users to create a virtual inventory of clothing items in their wardrobes. SCI provides real time usage tracking with Near Field Communication (NFC) tags attached to articles. NFC technology on smartphones makes it easier for the user to mark usage by simply scanning the article's tag. SCI not only keeps track of when an item was last used, but also how many times it has been used. It provides the ability to search articles based on names, tags, category, and usage filters. Additionally, it allows the user to match articles based on color to improve clothing choices and reuse of rarely used items. SCI web store allows users to share articles with other users and automate online advertising and selling of infrequently used, second-hand clothing items.

This report describes SCI, including the Android app, online web store, and the backend servers. First we discuss the NFC technology and how it works, followed by the SCI app features and use cases. Next, we discuss the technology stack, including NFC technology and tags, the image processing libraries used, design architecture of the server, the Android app, and the web store. Furthermore, we explore the design and implementation details of the Android app and web store. We conclude with summary of lessons learned, ideas for future and discussion of related work.

Table of Contents

List of Tables

List of Figures

INTRODUCTION

Humans are physical - we live and breathe in a physical environment. However, our lives are not limited to the physical environment around us. In this day and age, humans have social presences on the Internet via many services like Facebook and Twitter. With the advent of Internet of Things (IoT), more and more physical objects and devices are able to connect to Internet. IoT is changing and improving the way humans communicate with each other and the world around them. By 2020, Cisco [1] estimates the IoT will consist of 50 billion devices connected to the Internet. World population is expected to reach 7.7 billion people by 2020 [2]. These numbers are feasible because the IoT connects devices to other devices, as well as devices that connect humans to the Internet. For example, a Fitbit [3] and smartwatches track activity, exercise, food, weight, sleep and maps it. Vessyl [4] is a product promised to automatically track personal hydration needs. iLuminate [5] connect dancers wirelessly on stage, and enables them to turn the wearable lights on and off in time with the dancing and music.

The Internet of Things offers limitless options to connect human life to the Internet. With a single touch, we can find out what we ate, how many calories we burned, where we travel and more. This report focuses on an NFC based Android app to create inventory and track the real time usage of clothes worn by an individual. NFC tags are inexpensive, and NFC technology is an easy and fast way to enable devices in close proximity to communicate, without requiring an Internet connection [6]. Almost all smartphones today are enabled with NFC technology.

The SCI app running on an Android device acts as the reader. NFC tags attached to the clothes acts as other part of a wireless link. Usage is marked when the Android

device detects a tag. Users can further improve organization and clothing choices by matching tagged clothing items based on colors with image processing capabilities.

This report discusses how SCI enhances our physical world by leveraging the digital one. This section discusses the details of NFC technology and its use in SCI to enable digitization of closet inventory management. Chapter 2 gives detailed overview of all the technologies involved in SCI including, cloud services, algorithms, and mobile technology.

Chapter 3 covers SCI architecture for the backend server, image processing server, web client and the Android app. Chapter 4, Results, discusses SCI Android app UI flow and screenshots. Implementation Notes in Chapter 5 gives development environments used for SCI server and app and their respective code size along with some software metrics of libraries deployed on App Engine. Integration testing for SCI APIs is covered in Chapter 6. Finally, report concludes with a discussion section, with future enhancements to SCI and existing systems.

Chapter 1: Smart Closet Inventory (SCI)

This report presents SCI, an Android app for organizing individuals closet. The main set of features include:

- Virtual inventory with quick access to all the articles in the user's closet, sorted by categories
- Ability to read and write NFC tags attached to clothing articles for tracking usage
- Improved organization by tracking infrequently used articles by usage filters
- Improved clothing choices by matching articles based on colors

USES CASES & FEATURES

This section lists some of the real world uses cases of the two main features of SCI, NFC tagging and color matching.

Tracking Clothes usage over time

Clothes require constant organization. Some clothes are worn frequently but others are used rarely. SCI brings all the clothing articles in the closet into an app running on a smartphone or tablet. Next time the user is organizing their closet; they quickly filter all the unused items in the closet.

Use Case

Leila is running out of space in her closet. She wants to organize and figure out which articles have not been used in over a year so that she can get rid of these and make some space in her closet. Luckily, Leila has been using SCI app and has tagged all her clothes with NFC tags. She has been tagging articles from her closet every time she has worn them by simply tapping them to the back of her smartphone.

Over the weekend when Leila has some time to organize her closet, she launches the SCI app and selects the Search feature. She then sets the usage filter to one year, and SCI app lists all the articles that have not been used in last year. She can click on an individual article to view the image and check the article's purchase price. She uses this information to decide if she still wants to keep it.

Matching articles

The matching feature allows the user to search for matching color items given an article. When an article is created in SCI, the user is prompted to add an image. The image is used to extract the three most dominant colors in the article. This information is saved along with the article image. When looking for articles with matching colors, the user can tag the article, and SCI app will find the closest article based on color in a selected category.

Use Case

Arya has over 150 scarves, but she finds herself using a few selected ones more often. Two months ago she started using the SCI app and has tagged all the articles in her closet. She is planning a family trip to New York for the weekend and is packing her

outfits. She picks a blue dress, but when it comes to selecting a scarf, she picks a plain blue scarf. Then she remembers the SCI app's color matching feature.

Arya launches the app in Match mode and scans the NFC tag attached to her blue dress. The app instantly pulls up the article and displays the picture. Arya selects scarves as the matching category and touches the Match button. The SCI app lists all the scarves from the virtual closet that have the blue color in it. She realizes that she has a blue scarf with a red pattern, so she picks this scarf instead of the plain blue one, and later matches it with red shoes.

DESIGN AND IMPLEMENTATION

This section gives overview of the technologies used in SCI in Chapter 2. Chapter 3, System Architecture, discuss the design of SCI, including the server, web store, image processing services, and Android App. Integration with Android NFC and Google Sign-In services are also discussed. The Android app screen shots for all the features are listed in the Chapter 4. Chapter 5 and 6 discuss implementation notes and testing respectively.

Chapter 2: Technology Stack Overview

The SCI is designed using a number of technologies, frameworks, libraries, and algorithms to provide web store and an Android app. The web store and web APIs are hosted on Google App Engine with exception of the Image Processing service, which is hosted on Amazon Web Services. The Android app is integrated with web APIs hosted on Google App Engine and Amazon Web Services. The backend services require Google Sign-In ID token for user authentication. This section describes the dependent technologies of SCI in detail.

SECTION 2.1: NEAR FIELD COMMUNICATION (NFC)

Near Field Communication (NFC) has become an increasing growth area for research, retail, and consumers in IoT field. *The Internet of Things: A survey* [7], in 2010 recognized NFC and Wireless Sensor and Actuator Networks (WSAN) together with RFID as the atomic components linking the real world with the digital world. In *TMT Predictions 2015* [8], Deloitte predicts that by the end of 2015, "five percent of the base of 600-650 million near-field communication (NFC) equipped phones will be used at least once a month to make contactless in-store payments".

NFC technology is becoming increasingly popular. Applications include bank transactions, NFC ticketing [9], transferring data between devices, enabling/disabling smartphone features (e.g., turning off WiFi), and establishing a peer-to-peer (P2P) network which can be configured to use other wireless connections (e.g., connecting Bluetooth headphones to a smartphone).

NFC is a short-range wireless connectivity standard (ISO/IEC 18092) [10] and is based on RFID standard (ISO/EIC 14443). It uses magnetic field induction to enable communication between devices in close proximity, up to approximately 10 cm [11]. NFC standard operates at 13.56 MHz and supports data transmission rates such as 106 kbps, 212 kbps, and 424 kbps [11].

There are two types of NFC devices - passive and active. Active devices generate their own RF field, which in turn powers passive devices that do not have their own RF field. Passive devices, usually known as tags, are similar to RFID tags, containing an antenna and memory that could be read only, re-writable, or writable once. Active devices, such as NFC enabled smartphones, can switch between active and passive mode. In active communication both devices generate their own RF field while in passive mode only initiator generates the RF field to transmit the data.

NFC Forum has standardized format, NFC Data Exchange Format (NDEF) for data exchange. NDEF is a binary format structured in *messages*. Each of these messages contains several *records*, made up of a *header* and *payload*. Header contains metadata describing how to interpret the payload. The payload is the actual content [12].

NFC technology comes with some threats including but not limited to eavesdropping, data corruption, data modification, data insertion, and man-in-the-middle attack. Damme and Wouters explain how short range communication technology of NFC makes most types of attacks are practically impossible to mount. Haselsteiner and Breitfub [11] explain that man-in-the-middle-attacks is practically infeasible in real-world scenarios. NFC devices transmit data over radio frequency field and can detect an attack if the received signal does not match with the transmitted signal.

SCI does not write any personal information on the article tags. It only write article ID that is a UUID string. This string by itself does not give a lot of information. Moreover, SCI does not store article information on the device. It is always pulled from the server which authenticates each read and use request.

SECTION 2.2: NFC TAGS

SCI relays on tags attached to clothes for tracking real time usage. Clothes are exposed to relatively harsh conditions in laundry. In addition to water, they are exposed to chemicals and excessive heat. NFC tags used on clothes needs to withstand these conditions. For SCI, some of the researched laundry NFC tags are listed below:

- **Type 2 NFC Tag by tagstand** [13]: This tag is a 30mm circle with a hole design allowing easy sewing. It can store up to 144 bytes of user data. It can be made read-only, therefore, blocking data modification threats. Tagstand [13] claims that this tag has been test in the laundry and can withstand temperatures between -40 and 200 degree Fahrenheit.
- **NFC Laundry Token Advanced**: This is also a Type 2 NFC tag. It is waterproof and heatproof. It can withstand temperatures up to 200 degrees Celsius for 30 seconds [14].
- **NFC Laundry Token – NTAG213**: This tag can be custom ordered in range of 9mm to 30mm diameter. The NXP NTAG213 NFC chip is embedded in a black PPS material, which is waterproof and can withstand high temperatures up to 392 degrees Fahrenheit [15].

SECTION 2.3: GOOGLE APP ENGINE

SCI will be required to store user data for a large number of client apps. Once the user has created a closet from one Android device, they can simply login and retrieve the data from another device. Hence, SCI services must synchronize data across multiple client devices as well as the web store. SCI services and web store is hosted on Google App Engine. Google App Engine is a Platform as a Service providing easy development, deployment, and maintenance for the server. It provides scalability with Datastore services, and eliminates the requirement to maintain servers.

Smart Closet Inventory Services

SCI services are implemented in Python. Google App Engine provides the Python interpreter with standard Python libraries. It executes Python application code in a safe "sandboxed" environment [16]. Google App Engine provides only pure Python; hence no C libraries are supported. Image processing services of SCI use GraphicsMagick [17] a C library and are deployed AWS Elastic Beanstalk. The details for image processing services follow in Section 3.6.

SCI services are implemented in Python request handler classes using webapp2 framework. These request handlers read the requests in JSON format, process it, and build the responses in JSON format.

Datastore

In order to track real time usage of items, they need to be stored on the server and available for quick access when an NFC tag is scanned on a mobile device. The SCI matching feature also requires searching the entire closet inventory of a user to find

matching items. Future extensions for sharing closets with other users will require fast access and high availability of a user's closet data.

SCI utilizes NDB Datastore to store closet items and Blobstore to store the images of the items. App Engine provides Python APIs to directly access both NDB Datastore and Blobstore from services. NDB provides persistent storage in a schemaless object Datastore [18], which is ideal for storing simple closet items. Blobstore provides persistent storage for large files. This is ideal for storing closet items images. These images are used to extract the colors of the clothing items.

Front end web store

Google App Engine provides a simple web application framework, webapp2. A webapp2 application is divided into two components, `RequestHandler` and a WSGIApplication [19]. Smart Closet Inventory services are defined as `RequestHandler` classes in Python to process client request and return JSON responses. A WSGIApplication instance is used to route the incoming client request to these `RequestHandler` using URL mapping.

When webapp2 receives an HTTP GET request to a particular URL, it instantiates the handler for that class and invokes its `get()` method. The request JSON is parsed from `self.request`.

SECTION 2.4: ANDROID APP

One of the main features of SCI is to track real time usage of articles with NFC tags. This requires an NFC enabled active device to read and write the tags attached to

the clothing articles. NFC enabled smart device is the best available solution. This is the main reason to select Android platform for the SCI app. Google Nexus phones are NFC enabled since October 30, 2013, and Google Nexus tablets are NFC enabled since July, 2012 [20]. Since 2013 all major manufacturers have enabled NFC for Android devices. Moreover, the NFC feature is open on the Android platform, whereas on iOS the NFC technology is only available for Apple Pay. Therefore, iOS devices cannot be used to read or write NFC tags [21].

Android NFC Service

Android NFC powered devices are always looking for NFC tags when screen is unlocked, unless NFC feature is disabled. There are three modes of operation - Reader/writer mode, P2P mode, and Card emulation mode. SCI uses Reader/writer mode as it allows to read and/or write passive NFC tags. "The data stored in the tag can also be written in a variety of formats, but many of the Android framework API are based around a NFC Forum standard called NDEF (NFC Data Exchange Format)" [22].

SCI app wants to make the user experience seamlessly easier. It allows users to tag articles even when the SCI app is not running. This is achieved by filtering for ACTION_NDEF_DISCOVERED intent. Android provides tag dispatch system to handle tag detection. Android framework API allows applications, wanting to handle the NFC data, to filter for intent from the tag dispatch system. When a tag is detected, tag dispatch systems parses the NFC tag and encapsulates the MIME type or URI identifying the data payload into an intent. It then sends this intent to application that filtering. The Activity Chooser is launched when more than one application can handle the intent [23].

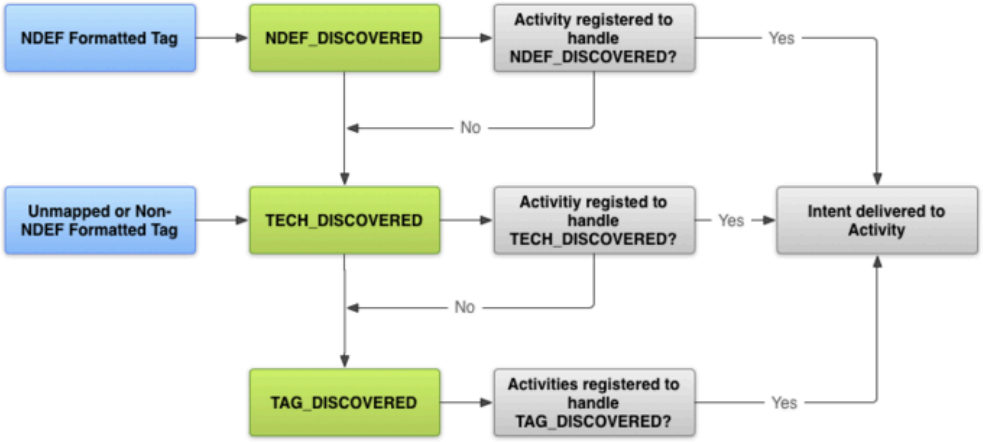

Figure 1: Tag Dispatch System

Figure 1 shows the flow for tag dispatch system with three intents. It starts an Activity with either ACTION_NDEF_DISCOVERED, the highest priority, or ACTION_TECH_DISCOVERED when parsing the NFC tag. If there are no activities filtering for these intents, it attempts to start an Activity with the next lowest priority intent that is either with ACTION_TECH_DISCOVERED or ACTION_TAG_DISCOVERED. This continues until an application filters for the intent or until the tag dispatch system has tried all possible intents. Finally, it does nothing if there's no application filtering for any of the three intents [23]. It is recommended to filter for the most specific intent, ACTION_NDEF_DISCOVERED.

Tag dispatch system repeats above process every time an NFC tag is scanned. This causes the Android to exit any current running app and launch the one handling the intent. This is the desired case but when user is already in the SCI app, it could lead to confusion if app kept restarting every time a tag is scanned. SCI app allows the user to mark the usage of the item at any screen in the app, with exception to matching and

search by tag screens. To overcome this problem, SCI uses a foreground dispatch system to handle NFC intents. This allows SCI app to intercept an NFC intent and claim priority over other activities or itself, if already running. Tag dispatch system is enabled when activity is running and disabled when activity moves to the background.

SCI has three different tag detection modes, read/usage mode, search mode, and match mode. Read/Usage mode is the default mode, supporting to mark usage of an item even without launching the SCI app. Search and match mode are explicitly enabled by the user when SCI app is running. Search mode allows searching for a particular article by its NFC tag. This speeds up the process of searching an item in the virtual inventory with by scanning the NFC tag. Match mode allows users to find the current article by scanning its tag and then finding articles with matching colors.

In addition to reading tags, SCI enables the user to write their own tags. SCI writes only unique article ID on a tag with which the item is identified on the server and finally in the Datastore. Users are prompted to write the NFC tag when creating a new article. After providing article name, price, category, tags, and privacy filter, article is created in the Datastore. Server returns unique article ID, which is then used to upload article image. Once article image is uploaded, user is prompted to scan a tag to write the article ID. SCI app enables write mode and looks for an NFC tag. When a tag is detected, foreground tag dispatch system makes sure that SCI app is not restarted and write mode remains enabled. SCI app handles the intent from the tag dispatch system and writes the tag with newly created article ID. Writing NFC tag requires the user to hold the tag against the android device rather than tapping. User is prompted on a successful write so the tag can be moved away from the device.

Google Sign-In

Another significant reason to choose the Android platform was the seamless integration and availability of Google Sign-In service. This eliminates not only the need for account creation and management on SCI server, but also provides authentication with the backend server to validate user session validity. SCI does not store any data on the device other than user profile and ID token. All data is pulled from the backend server using HTTP POST calls to the web services. Therefore, these requests have to be authenticated before returning any user data.

SECTION 2.5: IMAGE PROCESSING ALGORITHMS AND LIBRARIES

One of the main features of SCI is to match items based on colors. This is achieved by removing the background from the image and extracting the main colors. Then, this color data is stored for each of the items. SCI use several existing libraries to accomplish these three tasks including Lyst's [24] background removal approach, color survey [25], and Leifer's [26] color detection algorithm.

Background Removal

SCI uses pgmagick [27] for background removal using techniques mentioned by Lyst [24]. pgmagick is GraphicsMagick [17] binding for Python. The approach used by Lyst assumes that the item will be centered in the image, which is a fair assumption for SCI since the user will be focusing on the item when capturing the image. With this assumption, we create a mask from the original image and apply it in order to remove the background.

The first step is to determine the edges of the main item in the original image using pg-magick Sobel Filter [24]. Before applying Sobel Filter, colors are negated in the image using GraphicsMagick negate function [28]. This gives an inverted copy, which leaves a clear boundary for Sobel filter to later look for light to dark transitions [24]. This is achieved by GraphicMagick's edge function. The outliers left over from the sobel filter are smoothed out by Gaussian Blur, GraphicMagick's blur function, which blurs an image with specified blur factor [28]. Once the edges are detected a threshold function is applied to set all the non-black pixels to white. Then Local Adaptive Threshold is used to look for lone pixels in 5x5 areas and remove those that are not in a cluster [24]. This is achieved by GraphicMagick's adaptiveThreshold function, which "works by evaluating the mean (average) of a pixel region (size specified by width and height) and using the mean as the thresholding value" [28]. Finally, the thresholded image is flooded with magenta from the corners, and then magenta pixels are turned transparent to create the mask on the original image [24]. When this mask is applied on the original image, the background is removed. This process is shown in Figure 2 from [24].

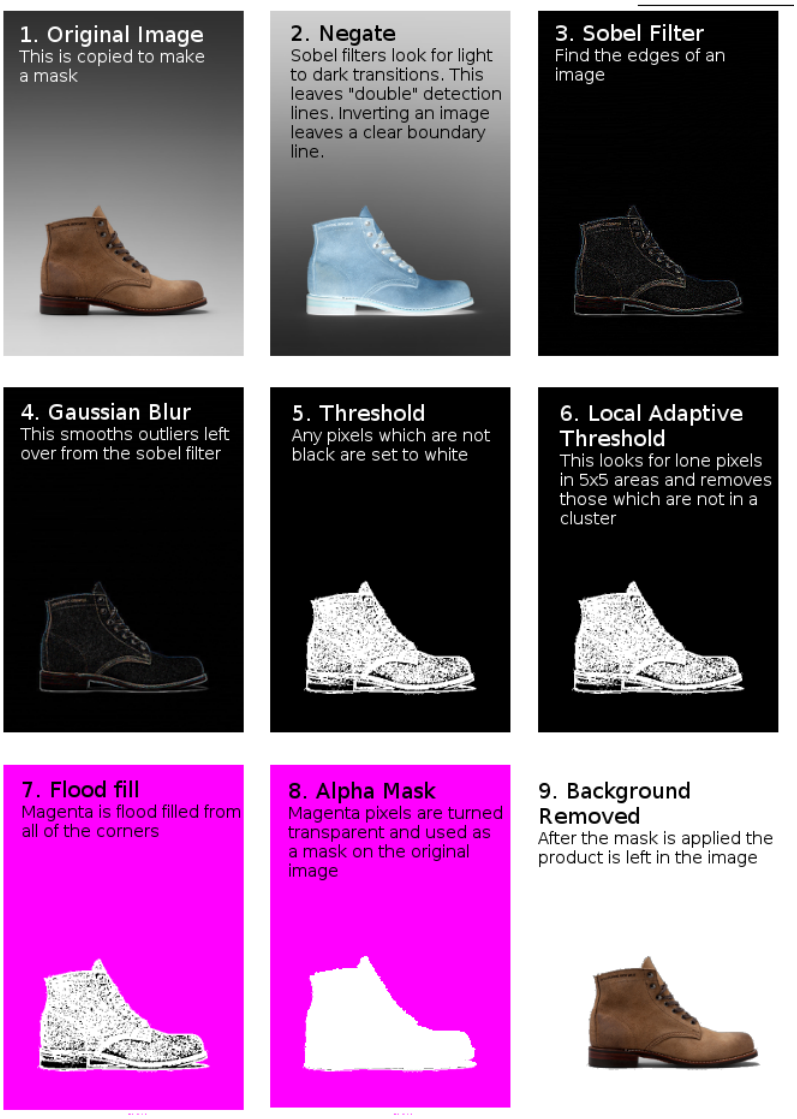

Figure 2: Background Removal Process

This algorithm gives very close results for determining the original image background, but it is not perfect. Therefore, SCI allows the user to adjust the colors for the item and use the automatic color detection as a starting point.

Color Extraction

SCI use k-means clustering approach proposed by [29] to extracts three dominant colors from the image giving three clusters. K-means is also used for color image segmentation, vector quantization and data compression [30]. Park *et. al* [31] proposes the use of k-means algorithm for clustering in pattern classification and image segmentation. Weeks and Hague [32] use k-means algorithm for color segmentation in the HSI color space.

K-means clustering algorithm knows in advance how many distinct clusters to generate. It determines the size of the clusters based on the structure of the data. It started with k randomly placed centroid, "points in space that represents the center of the cluster" [33]. It first assigns every item to the nearest one and then moves the centroids to the average location of all the associated points and redoing the assignments. This process is repeated until the assignments stop changing. The end result looks something like figure 3 [29] where the points are colored based on the cluster they belong to, and the dark-black circles indicate the centers of each cluster.

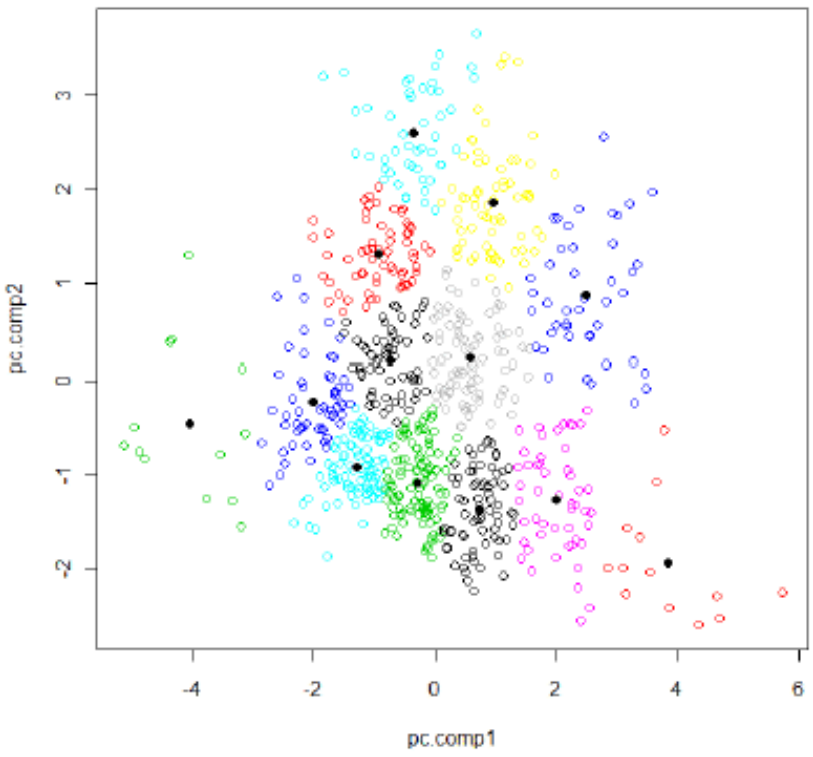

Figure 3: K-means

When applying k-means to the image, each pixel is treated as a point on a 3-dimensional space. Then all the pixels are clustered in groups based on color and their centers are selected as the dominant colors. [29] provides a few optimizations to resize the image down to 200x200. It also saves calculations by storing a count with each point instead of storing "duplicate" points.

This algorithm gives hexcode for the three dominant colors in the image. This introduces two difficulties:

- Hex color codes are difficult for humans to interpret.

- Hex color codes determine the colors to a very granular level. This makes it almost impossible to find matches of items in same color range, say purple.

Therefore, the next step in SCI image processing is to determine the name of the detected colors based on the closest range to a color family.

Color Name Identification

Color Survey by Munroe [25] is used to convert a hex color code to a human. This survey resulted in a list of 200k RGB color values translated into a small set of color names. Even with these modifications, 200k color values only covered around 1.2% of the RGB space (200000 / 255^3). Therefore, Lyst [26] suggests considering the distances between the colors to map all the hex color codes to color names.

Color Survey

This section covers details from Randall Munroe's survey in which over five million colors inside the RGB space of the average monitor were named across 222,500 user sessions [25]. Most of the data comes from "the sample of all non-colorblind users on all types of monitors (>90% LCS, roughly 6% CRT)" [25].

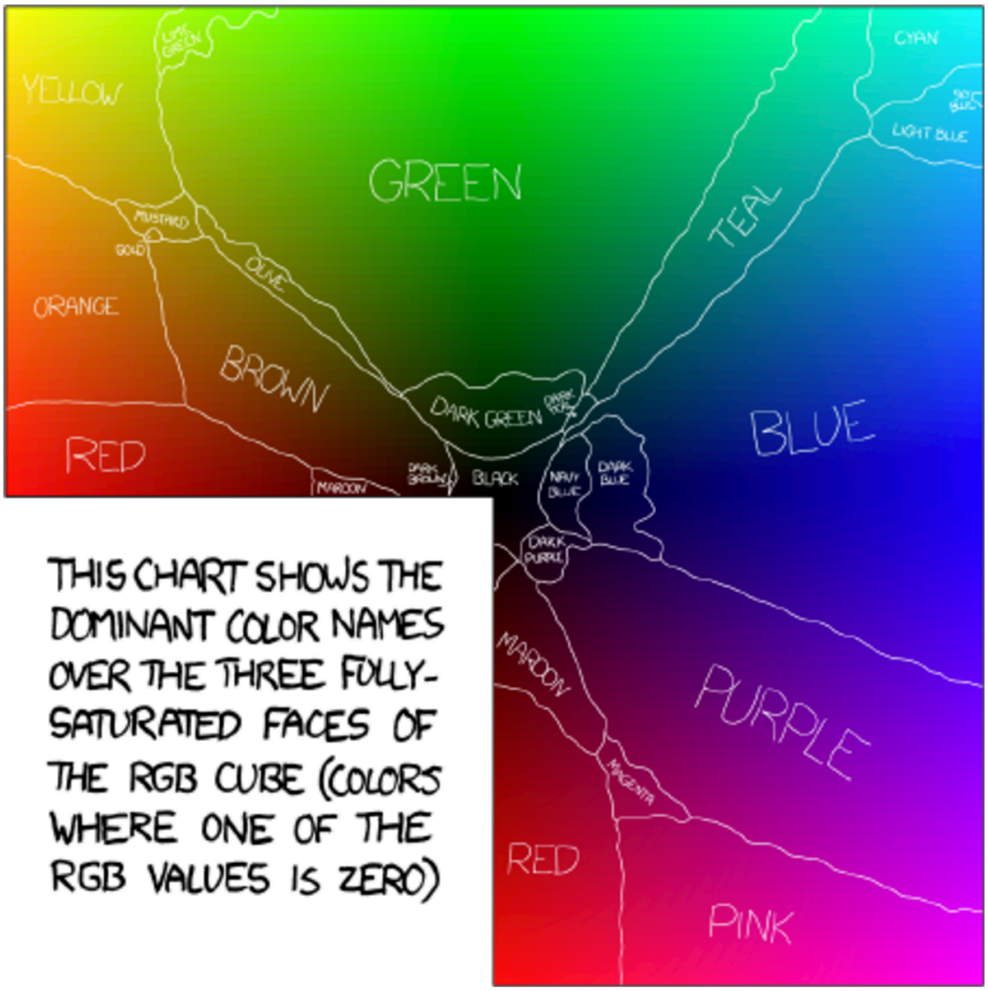

Figure 4: Map of color boundaries

Munroe [25] used the average of several runs of a stochastic hillclimbing algorithm to identify the RGB value for a name. This RGB value "is based on the location in the RGB color space where there was the highest frequency of response choosing that name" [25]. The complete list of 954 colors is listed in [34]. Figure 4 [25] shows the map of color boundaries for a particular part of the saturated faces of the RGB cube.

Color Distance

xkcd data for color names does not cover all the colors in RGB space. To cover all the hex codes we use the Color Distance approach by Lyst [26]. The idea is to find similar colors based on the distance from the named colors and name them accordingly. The difficulty is that "RGB space is not suited to measure color difference because distance magnitudes in the color space do not necessarily correspond to the magnitude of color difference as perceived by humans." [26]. Lab color space [35] provides a solution to this problem.

The Lab Color model defines the colors on a three-axis color system where colors are absolute and therefore device independent. The intention behind color space is to create a space that is more perceptually uniform than XYZ but can be easily computed. Perceptual uniformity is defined as "a change of the same amount in a color value should produce a change of about the same visual importance" [35]. The three axes include *L* for lightness and *a* and *b* indicating the color dimension. The *L* axis consists of white, black and gray colors and goes up and down the Lab Color model. The *a* axis goes from cyan color to magenta/red color, while *b* axis goes from blue to yellow [36]. Lab color space includes all the perceivable colors with gamut exceeding those of the RGB and CMYK color models [35].

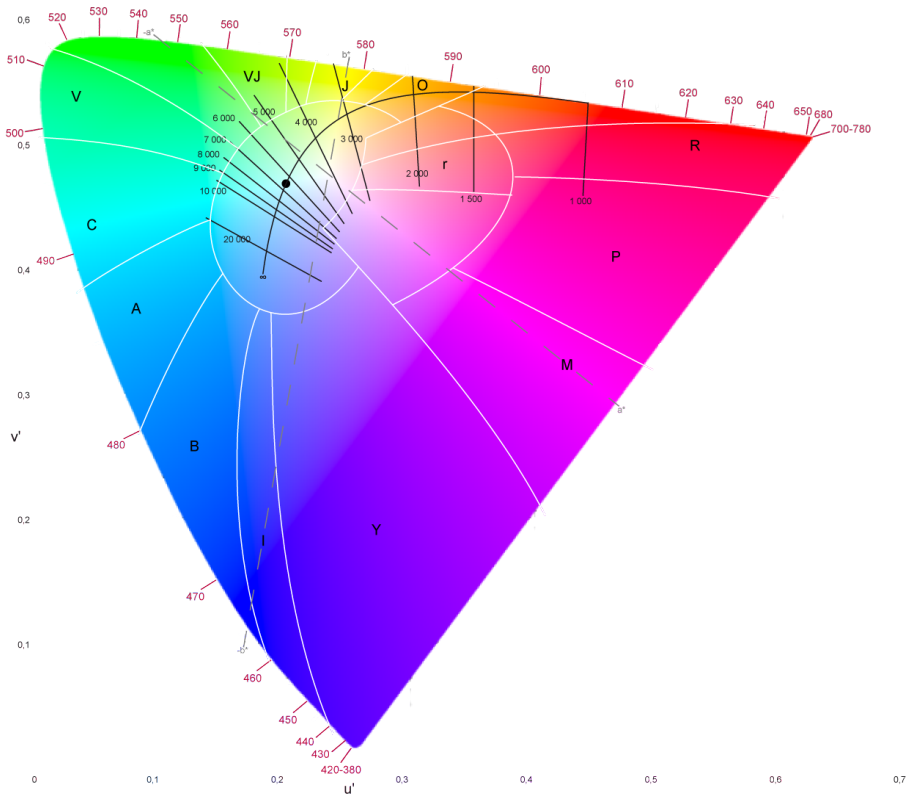

Figure 5: Lab Color Model

International Commission on Illumination's (CIE) first representation of Color space was CIE 1931. SCI is using deltaE function in CIE 2000 color space. Following the original approach of Lyst, SCI use vectorised deltaE to speed up the color difference calculations over 200k colors in xkcd matrix. The vectorised deltaE functions [37] are available in the colormath package [38].

First the 1000 colors are loaded from xkcd as numpy array. Then, we get the associated RGB color from the hex code of the color we want to name. Using RGB color, we create lab color, which is later converted into a lab color numpy array. Next, vectorised deltaE function is used to find the closest match to the lab color from the xkcd

lab matrix. Once the closest match is identified, its index is used to lookup the actual color name.

```
import os
import os.path
import cPickle as pickle
import numpy as np

import colormath
from colormath.color_diff_matrix import delta_e_cie2000
from colormath.color_objects import LabColor, sRGBColor, XYZColor, LabColor
from colormath.color_conversions import convert_color

lab_colors_path = os.path.join(os.path.dirname(__file__), 'pickle', 'lab-colors.pk')
color_reader = pickle.load(open(lab_colors_path, "rb"))

# Load 1000 colors from the xkcd
lab_matrix_path = os.path.join(os.path.dirname(__file__), 'pickle', 'lab-matrix.pk')
lab_matrix = np.load(lab_matrix_path)
```

```
 def get_color_name(self, hexcolor):
   # get the rgb color from the hexcolor code
   rgb = sRGBColor.new_from_rgb_hex(hexcolor)

   # conver rbg color to lab color
   lab = convert_color(rgb, LabColor)

   brgb = convert_color(lab, sRGBColor)

   # create a lab color vector based on l, a, and b axiss
   lab_color_vector = np.array([lab.lab_l, lab.lab_a, lab.lab_b])

   # find the closest match to the lab color in the 1000 xkcd lab_matrix
   delta = delta_e_cie2000(lab_color_vector, self.lab_matrix)

   # get the index of the closest match
   index = np.where(self.lab_matrix == self.lab_matrix[np.argmin(delta)])[0][0]

   # return the name of the closest matched color
   return self.color_reader[index]
```

Figure 6: Color name identification algorithm

SECTION 2.6: AMAZON WEB SERVICES

SCI use Lyst's [24] proposed solution for the image background removal, which required using pgmagick [27] and GraphicsMagick [17] libraries. GraphicsMagick [17] could not be deployed on Google App Engine because it is written in C. We investigated alternate Python solutions, using Python Image Library (PIL) but either the results were not accurate or the image processing requests timed out. Next, we considered AWS Elastic Beanstalk which allows virtual deployment of Linux environments in addition to other operating systems. This allows installation of GraphicsMagick [17] and pgmagick [27] libraries available for Linux. Therefore, instead of reinventing the image processing capabilities in pure Python, SCI deploys image processing services with existing C libraries on AWS Elastic Beanstalk.

Chapter 3: System Architecture

Section 3.1: Overview

The SCI uses client-server architecture. The server is hosted on Google App Engine and accepts requests through HTTP. All requests are authenticated based on Google Sign-In ID token. In addition, image processing service is deployed as a Django [39] app, again accepting requests through HTTP on Amazon Web Services. The following section discusses all the components of the architecture in detail.

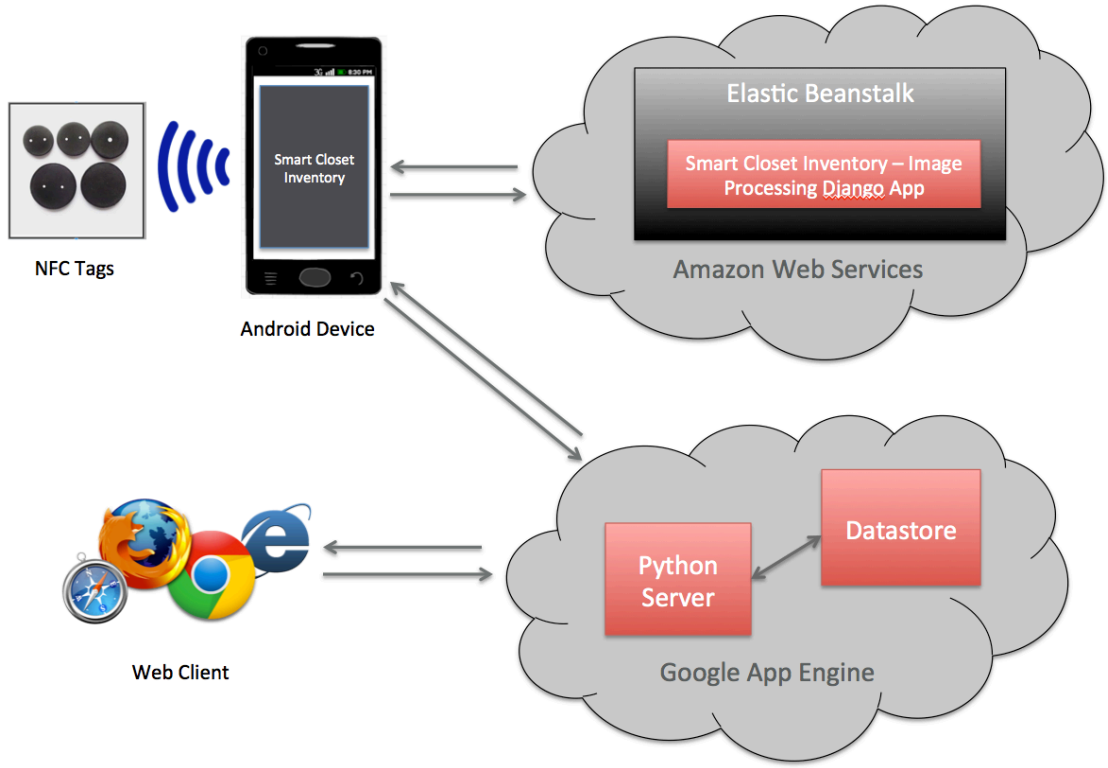

Figure 7: Smart Closet Inventory System Architecture

SECTION 3.2: APP ENGINE SERVER

This section covers detail of Python server and Datastore.

Python Server

SCI server is a webapp2 application hosted on App Engine in Python Runtime Environment. All services are implemented as webapp2 application RequestHandler. When a request is received, the appropriate RequestHandler service class parses the JSON, processes the request, and then returns the response as another JSON. The complete list of SCI service RequestHandler is given below:

Token Sign in Handler

This service handler is called when the currently signed in user account needs to be authenticated. This service takes user's ID token as JSON parameter. After the user signs in (using Google Sign-In) on the client side, the user's ID token is retrieved. This ID Token is then sent to this service using HTTPS. This service validates the ID token using the tokenInfo endpoint. It makes an HTTPS POST request with ID token as `id_token` parameter. If the token is not correctly signed, then an error is returned. Otherwise, an HTTP 200 response is returned with a JSON-formatted ID token claims [40]. Once the claims are returned, Token Sign in service validates that it was issued to one of the app's client ID and returns a successful response for authentication.

Create Profile Handler

This service is called in order to create a new user account. It requires two parameters, username and email, and creates an account for the new user in the Datastore.

Create Article Handler

This service is called when a user wants to add a new article to the inventory. It authenticates the ID token. If valid, it creates an article in the Datastore. It returns the string UUID of the newly created article, and takes basic article information as listed below along with `tokenId`:

- Name
- Description
- Price
- OK to sell flag
- Tags
- Owner
- Private flag
- Type, indicating which category this article belongs to

Android Image Upload Handler

This service is called when the user adds an image to a newly created article. It requires ID token, article ID, and the actual image data. If the ID token is valid and the article exists in the Datastore, it creates an image file in the Blobstore. The article is updated with the image URL from the Blobstore, and this URL is returned with the success code.

Update Article Image Colors Handler

This service is called to update the article with the three dominant colors extracted from its image. It takes three hex color codes and the article ID. If the article exists, it uses the color identification algorithm to identify the color names for each of the

hex code. Finally, it updates the article in the Datastore with the color names. It returns success or failure based on the service execution.

Update Article Handler

This service is called when any of the basic information needs to be updated for a given article. It takes ID token, article ID, field to be updated, and the append flag. The append flag applies to fields containing lists, such as article colors or tags. If the ID token is valid and the article exists, the service updates the appropriate article field in the Datastore. It returns a success or error code and a list of fields updated.

Read Article Handler

This service is called to load the article with a given article ID. It is usually called when an article is tagged for matching from the Android app client. It takes ID token, article ID, and email of the logged in user. If the token is valid and the article exists, the server returns the article information to the client in JSON-formatted response, along with success code. Otherwise an error is returned.

Use Article Handler

SCI only supports marking usage by tags for now. This service is called when an article is tagged to mark the usage from Android app client. It takes ID token and article ID. If the ID token is valid and article exists, the server updates the article with the current date as the usage date and increments the number of times the given article has been used so far. It returns either a success or error depending on the Datastore update operation.

Search Article Handler

This service is called when searching articles by name, tags, description, color, usage filter, tag, never used, or selling filter. It takes ID token, filter type (article field), filter string, and email. If ID token is valid, this service search for the matching articles and return in JSON-formatted response if owned by logged in user.

Get Categories Handler

This service is called when retrieving the entire closet for a given user. It takes ID token and email filter. If the ID token is valid, it retrieves all most used articles owned by the logged in user and sorts them by type, indicating the categories (coats, scarves and so on). It returns a JSON-formatted list of `currentCategories`, each with the image of most the used article of that category.

Get Category Handler

This service is called when retrieving all the items in a selected category owned by the logged in user. It takes ID token, category, and email filter. If the ID token is valid, it returns a JSON-formatted list, category, containing all the articles owned by the logged in user.

Find Match Handler

This service is called when finding a match given an article. It takes ID token, article ID, email, and category to match. If ID token is valid, it retrieves the original article from the Datastore using article ID. Next, all the items from the selected category owned by the logged in user are retrieved. Finally, colors for each item in the selected category are matched with the original article, and it is added to the return response list if

even one color match is found. This service returns a JSON-formatted list, `articleList`, as the response.

Datastore

SCI app utilizes NDB Datastore to store closet items and Blobstore for storing images for each item. NDB Datastore provides persistent storage in a schemaless object Datastore [18]. Each data object is known as an *entity*. Entity has one or more *properties* to store the data values. Data values are one of the supported NDB data types. SCI consists of three main entities - *User*, *Article*, and *ArticleImage* shown in figure 8.

Each user has an entity that stores the user name and email. SCI uses Google Sign-In library, hence there is no need to store the user's password. Users are authenticated using ID tokens generated at the time of user login. The *Article* entity stores all the information about the article, including the image URL. The *ArticleImage* holds the image details and the article ID to which the image is associated. Each article has one image. Images are stored as *blobs* using Blobstore API, which creates an upload URL. This URL is stored in the *Article* and *ArticleImage*.

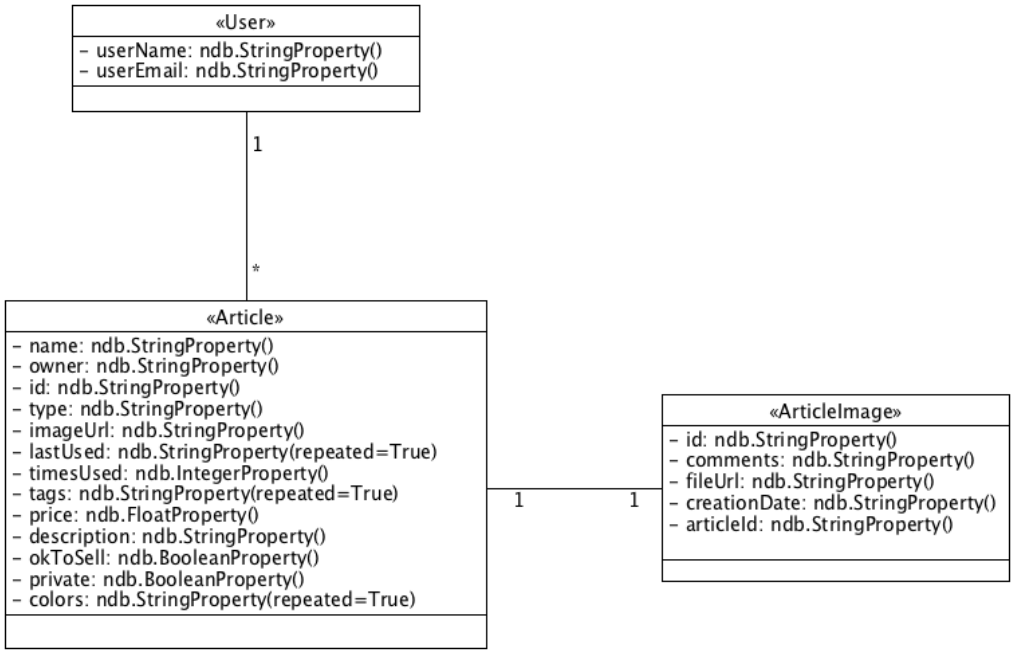

Figure 8: SCI server entities UML Diagram

SECTION 3.3: AMAZON WEB SERVICES

Google App Engine provides only pure Python and therefore no C libraries are supported. SCI depends on pgmagick, [27] which is binding for a C library, GraphicsMagick [17]. Therefore, I wrote a service which performs background removal. This service is deployed on Amazon Elastic Beanstalk. In addition, the color extraction service is also deployed on Amazon Elastic Beanstalk to eliminate the unnecessary requirement of porting back the intermediate image without the background to Google App Engine server for further processing. The image processing service is deployed on a simple Django Web application [39].

The Elastic Beanstalk environment can be customized with third party libraries at the time of deployment using a configuration file. SCI configuration file include the

commands to install GraphicsMagick, pgmagick, and PIL libraries packages, and then link them with the Django runtime Python libraries.

The image processing service takes an image URL, loads the image with pgmagick, and removes the background. The image is then stored at a temporary location on Elastic Beanstalk and is used for color extraction. Once three dominant colors are extracted, the hex code for each color is returned as a JSON-formatted response.

SECTION 3.4: WEB CLIENT

The SCI comes with a front end web store to share articles and list them for sale. In the current implementation, the automatic sale feature is not yet enabled. The web store is designed using Start Bootstrap [41] template, Shop Item bootstrap [42]. Shop Item template is an unstyled template to list items for sharing and sale, which is perfect for SCI. It features Bootstrap snippets from Bootsnipp [43].

The Shop Item template is branded with Smart Closet Inventory and is deployed using webapp2 framework on Google App Engine. Details of SCI server are discussed in Chapter 3.2 - App Engine Serve, Python server.

SECTION 3.5: ANDROID APP

In this section we discuss the detail implementations for the SCI app. This section is divided into four sections each covering four main functionalities of the app. These sections discuss UI design and implementation, communication with backend services (including Google App Engine and Amazon Web Services), Google Sign-In and NFC Tagging.

UI Design and Implementation

In Android, an application starts with a main activity. When an activity is first created, the first method invoked is *onCreate()*. Android activities tend to be single focused and interacting with user.

Fragment represents a portion of the user interface in an activity. When an activity is paused, all fragments in it are also paused. When an activity is destroyed, all fragments in it are also destroyed [44].

SCI app uses only one *activity*. The rest of the app screens are implemented as *fragments*. This design is useful for enabling NFC tagging on multiple screens of the app, like marking article usage, searching using an article tag, or finding matching items for a tag. SCI's main activity is always looking for handling NFC intents. Whenever a NFC tag is tagged from a fragment, it returns and is handled in the main activity.

Figure 9 from [45] shows the important state paths of an Android activity. The major states of an activity are represented by colored ovals. The callback methods are represented by square rectangles.

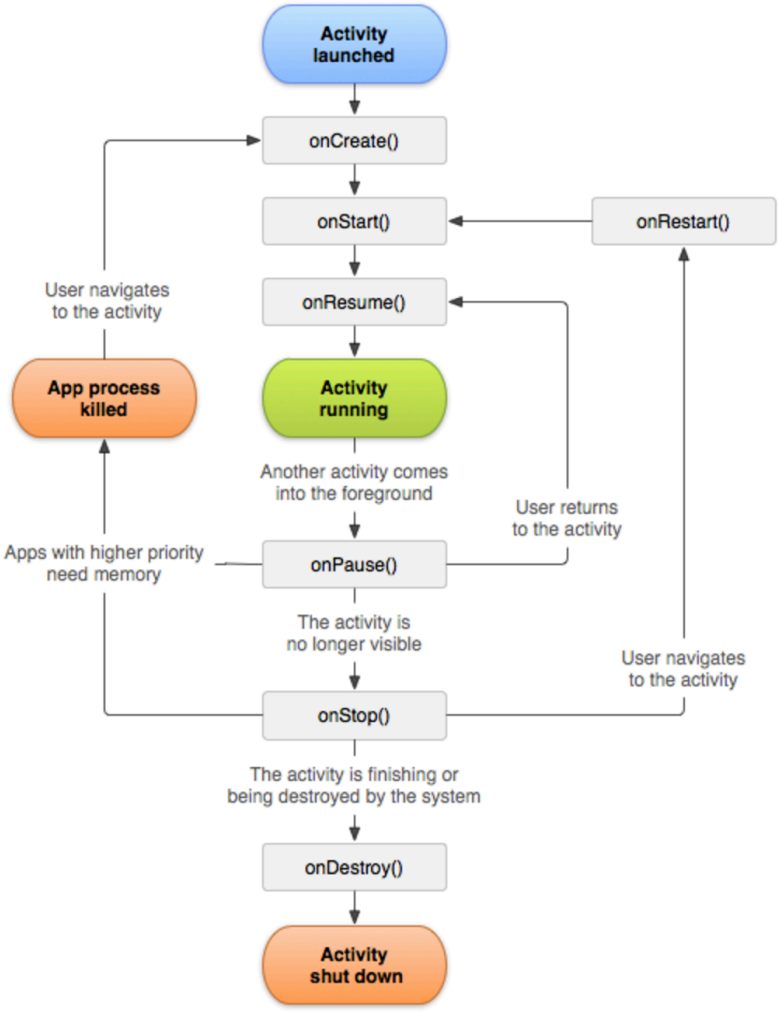

Figure 9: Android Activity Lifecycle

The *onCreate()* in SCI's main activity is overridden to perform the following three functions:

- Initialize NFC adapter for tag detection
- Load Slider menu for the app screens
- Authenticate the logged in user

If user authentication fails, app is redirected to the sign up screen. Upon successful login, Home screen is loaded. In addition to this basic flow, SCI's main activity implements fragment interaction listeners for each of the SCI fragment. This allows main activity to manage the flow between the fragments as each fragment calls back the main activity once it is done with its work.

Client-server communication

All the client-server communication is handled using Android Intent Service and Broadcast Receiver. Android Intent Service is used to handle asynchronous requests. These requests are expressed as Intent and started as needed using `startService(Intent)`. Request Intents can be started from activities or fragments. When Intent Service receives a request, it launches a worker thread to handle it and stop the service when it runs out of work [46]. The main UI flow remains responsive while Intent Service handles the asynchronous request. Any data can be passed to Intent Service using extras.

When SCI app needs to invoke a backend server call, main activity and each of the other fragments invoke Smart Closet Intent Service, which extends Android Intent Service. This service makes HttpPost object and executes the POST request. In order to create HttpPost object, the request URL and appropriate request JSON is required to identify which service to call. The client activity or fragment invokes the service and adds a REQUEST_URL and REQUEST_JSON as extras. After that client activity forgets it.

Once invoked, `onHandleIntent(Intent)` method of Smart Closet Intent Service is called where REQUEST_URL and REQUEST_JSON are extracted from

extras and added to an HttpPost object. Then, this service executes the POST request and gets the response.

Smart Closet Intent Service needs to then send back the response to the calling activity or fragment. In order to achieve this, it sends back a broadcast from the `onHandleIntent()` method to inform any Broadcast Receiver of the results. SCI's main activity and all other fragments define a Broadcast Receiver subclass and register the receiver [47]. When a broadcast is sent across applications, the registered receiver gets it and processes the JSON response from SCI server in its `onReceive()` method.

SCI servers running on both Google App Engine and Amazon Web Service return JSON-formatted responses. These responses are converted into Java Objects using Gson Java library using a JSON parser utility [48]. In Android, SCI app defines three main types of data objects - *Article*, *Category*, and *User*. An additional parser is used to parse a JSON array to *Article* Java objects, as most of the SCI web services return a JSON-formatted list of articles.

Integrating Android NFC service into Android app

In order to use NFC on Android device, `android.permission.NFC` and hardware permission is required to be added in the AndroidManifest.xml, shown in figure 10.

```
<uses-permission android:name="android.permission.NFC" />

<uses-feature
    android:name="android.hardware.nfc"
    android:required="true" />
```

Figure 10: Requesting NFC Access

Filtering for NFC Intents

SCI filters for ACTION_NDEF_DISCOVERED intents. Figure 11 shows this is achieved by declaring intent filter in the AndroidManifest.xml with MIME type of text/plain.

```
<intent-filter>

    <!-- To launch the NFCApp automatically, since NDEF_DISCOVERED
    has the highest priority -->
    <action android:name="android.nfc.action.NDEF_DISCOVERED" />

    <category android:name="android.intent.category.DEFAULT" />

    <data android:mimeType="text/plain" />
</intent-filter>

<meta-data
    android:name="android.nfc.action.NDEF_DISCOVERED"
    android:resource="@xml/nfc_tech_filter" />
```

Figure 11: Filtering for NFC intent

Handling NFC intent

SCI uses a foreground dispatch system to handle NFC intents. The foreground dispatch system makes sure that an Android system does not exit the app and open another one to process a new NFC tag detection [49].

To enable the foreground dispatch system, a `PendingIntent` object is required. This is populated by the Android system with the details of the tag when it is scanned. The example code is shown in figure 12.

```
protected NfcAdapter nfcAdapter;
protected PendingIntent pendingIntent;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    //initialize NfcAdapter to start tag detection
    initializeNfcAdapter();
    handleNfcIntent(getIntent());

    //load slider menu items
    loadSliderMenu(savedInstanceState);
}

//--------------- NFC Processing Methods ---------------

private void initializeNfcAdapter() {
    nfcAdapter = NfcAdapter.getDefaultAdapter(this);
    pendingIntent = PendingIntent.getActivity(this, 0,
        new Intent(this, ((Object) this).getClass())
        .addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP), 0);

    if (nfcAdapter == null) {
        ToastMessage.displayShortToastMessage(this,
        "NFC is not supported on this device.");
        finish();
        return;
    }

    if (!nfcAdapter.isEnabled()) {
        ToastMessage.displayShortToastMessage(this,
        "NFC is disabled.");
    } else {
        ToastMessage.displayShortToastMessage(this,
        "NFC is enabled.");
    }
}
```

Figure 12: PendingIntent Object

Next, an intent filter is required to intercept and handle the intents. This is required because whenever a tag is scanned, the foreground dispatch system checks the specified intent filters with the intent received from the tag scan. If it matches, it is handled by the activity. If it does not match, then it falls back to the intent dispatch system [50]. SCI declares this intent filter in the `enableForegroundMode()` helper method.

```
@Override
protected void onPause() {
    super.onPause();
    disableForegroundMode();
}

@Override
protected void onResume() {
    super.onResume();
    enableForegroundMode();
}

public void enableForegroundMode() {
    IntentFilter tagDetected = new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
    IntentFilter[] filters = new IntentFilter[] { tagDetected };
    nfcAdapter.enableForegroundDispatch(this, pendingIntent, filters, null);
}

private void disableForegroundMode() {
    nfcAdapter.disableForegroundDispatch(this);
}
```

Figure 13: Enabling foreground dispatch system

The foreground dispatch system should be enabled only when an activity handling NFC tag is running and should be disabled when the activity goes into the background. In figure 13, SCI calls `enableForegroundDispatch()` in `onResume()`. Similarly it calls `disableForegroungDispatch()` in `onPause()`.

Finally, `onNewIntent()` callback is overridden to process the intent, shown in figure 14. If write mode is enabled, the tag is written with article ID. If write mode is disabled then tag is read by processing the data, article UUID.

```
/* OnNewIntent read the tag */
@Override
public void onNewIntent(Intent intent) {
    if(writeMode) {
        writeMode = false;
        Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
        writeTag(tag);
    } else {
        setIntent(intent);
        readTag(intent);
    }
}
```

Figure 14: Handling NFC intent

Once the article UUID is retrieved, a JSON request object is created with the article UUID, ID token, and logged in user email. This request JSON is then passed to the READ_ARTICLE web service.

- If match mode is enabled, then Request Receiver is set to launch the match article view.
- If search mode is enabled, then Request Receiver launches the search by tag view with article result.
- If read mode is enabled, then the Use Article web service is called instead of Read Article. The Request Receiver for User Article service does not refresh the app view as the article is marked used in the background.

Google Sign-In

SCI app uses Google Sign-In service to sign up or login users with their Google accounts. In order to integrate Google Sign-In service to Android, it requires following:

Configuration File

This file provides service-specific information for SCI app [51]. It is generated based on the SCI server app and Android app package. It also requires the SHA-1 of the signing certificate. This configuration file is then downloaded as JSON and added to app/mobile/ directory of the Android app project.

Google Services plugin and Google Play services

The configuration information from the `google-services.json` is parsed using Gradle, which required Google Services plugin. The plugin is added in the `build.gradle` files of top-level and app-level. In addition Google Services are also added as a dependency in the app-level `build.gradle` [51].

OAuth 2.0 client ID

OAuth 2.0 client ID is required only if we are going to authenticate with the backend server [51]. This model meets SCI client server authentication. Once a user has been logged in on the Android device, all the future requests can be authenticated with the backend server before it returns any data to the client.

Integrating Google Sign-In into Android app

This section describes how Google Sign-In is integrated into SCI app and authenticated with the backend server. Main activity in SCI app is responsible for the UI flow between all app fragments and the NFC service. Therefore, Google Sign-In is handled in a separate sign in fragment to simplify the app code design.

Google Sign-In is configured by creating `GoogleApiClient` with access to the basic profile in the `onCreate()` method of sign in fragment [52]. SCI app code is shown in figure 15.

```
//-- Google Signin ---
// Build GoogleApiClient with access to basic profile
mGoogleApiClient = new GoogleApiClient.Builder(getActivity())
        .addConnectionCallbacks(this)
        .addOnConnectionFailedListener(this)
        .addApi(Plus.API)
        .addScope(new Scope(Scopes.PROFILE))
        .build();
```

Figure 15: Initializing GoogleApiClient

Next sign up, sign in and logout buttons are customized as per Google branding and added in the sign in fragment layout. The only difference between sign up and sign in is that upon sign up an extra request is sent to the backend server to create the appropriate user account on SCI server.

Once the buttons are added, we set on-click listeners for each of the buttons. In the `onClick()` method of the sign in fragment we then call the appropriate handling method for clicking the sign up, login and logout button.

```
@Override
public void onClick(View v) {
    if (v.getId() == R.id.googleSignInButton) {
        onSignInClicked();
    } else if (v.getId() == R.id.googleSignUpButton) {
        onSignUpClicked();
    } else if (v.getId() == R.id.signoutButton) {
        logoutCurrentUser();
    }
}
```

Figure 16: Handling user selection for login and logout actions

Figure 16 shows when sign up is clicked, the sign up flag is set to true, indicating the creation of a new account, and then `GoogleApiClient`'s connect method is invoked. When sign in button is clicked, sign up flag is set to false before calling `GoogleApiClient`'s connect method.

`GoogleApiClient`'s connect method connects to Google Play services and returns immediately after connecting to the background service [53]. On successful connection, `onConnected(Bundle)` is called indicating that a Google account was selected and has granted requested permissions to the requesting app. Whereas on a connection failure, `onConnectionFailed(Bundle)` is called [53] indicating that connection with Google Play services failed, and the user either needs to select an account, grant permissions, or resolve any errors. SCI app's sign in fragment overrides both of these callback methods.

If sign in fails, the app is re-directed to the sign up view, prompting user to login. In `onConnected(Bundle)` method, user email and username is retrieved, which is only available if user has a Google+ account using `Plus.PeopleApi`. If sign up is enabled, sign in fragment then calls the backend server to add this user account. Next, sign in fragment reset the ID token for this account and creates a user profile by adding user name, email, and ID token to the Shared Preferences of the app. This is achieved in a background `AsyncTask`, and the app is redirected to the Home fragment.

```
//-- Google Signin ---
@Override
public void onConnected(Bundle bundle) {
    // onConnected indicates that an account was selected on the device,
    // that the selected account has granted any requested permissions
    // to our app and that we were able to establish a service connection
    //to Google Play services.
    mShouldResolve = false;

    //only works if user has a google+ profile
    if (Plus.PeopleApi.getCurrentPerson(mGoogleApiClient) != null) {
        Person currentPerson = Plus.PeopleApi.getCurrentPerson(mGoogleApiClient);
        userName = currentPerson.getDisplayName();
        String personPhoto = currentPerson.getImage().getUrl();
        String personGooglePlusProfile = currentPerson.getUrl();

        ToastMessage.displayLongToastMessage(getActivity(),
        "Logged in as : " + userName);
    } else {
        Log.e(SmartClosetConstants.SMARTCLOSET_DEBUG_TAG,
        CLASSNAME + ": User doesn't have a google plus profile...");
    }

    // retireve user's email address
    userEmail = Plus.AccountApi.getAccountName(mGoogleApiClient);

    if (signingUp) {
        // create account with SCI server
    }

    // reset user tokenId and set the user Profile
    new GetIdTokenTask(getActivity()).execute();

    // load HomeFragment
    if (!logout) {
        onSigninFragmentInteractionListener.
        onSigninFragmentInteraction(false, false);
    }
}
```

Figure 17: Retrieving Google account information

When logout button is clicked, `GoogleApiClient`'s `disconnect()` method is called to logout the user. The user profile and ID token is removed from the Shared Preferences, and sign in fragment callbacks the main activity to redirect to the signup view.

Authenticate with a backend server

SCI app communicates with the backend server deployed on Google App Engine. All user inventory data resides on this server, therefore any request made to access user

data needs to be authenticated with the SCI backend server. To authenticate in a secure manner, the user's ID token is retrieved upon successful login in the Android app. This ID token is then sent with every web service request to the SCI backend server, which validates the validity of the ID token and grants access to the user data.

Chapter 4: Results

This section discusses the SCI Android app UI flow and screens for all features. As mentioned earlier in Chapter 3: System Architecture, SCI app contains only one main activity which always looking for NFC tags in read mode. When a tag is detected, the app screen flow is not affected and SCI app silently marks the article usage in the background.

SECTION 4.1: ANDROID APP SCREENS

Figure 19 shows the Home screen. If user authentication is successful, the Home fragment is loaded to mark the usage of the item with tagging enabled. If user authentication fails, the user is redirected to the sign in fragment. All links of the slider menu are redirected to the sign in fragment when authentication fails.

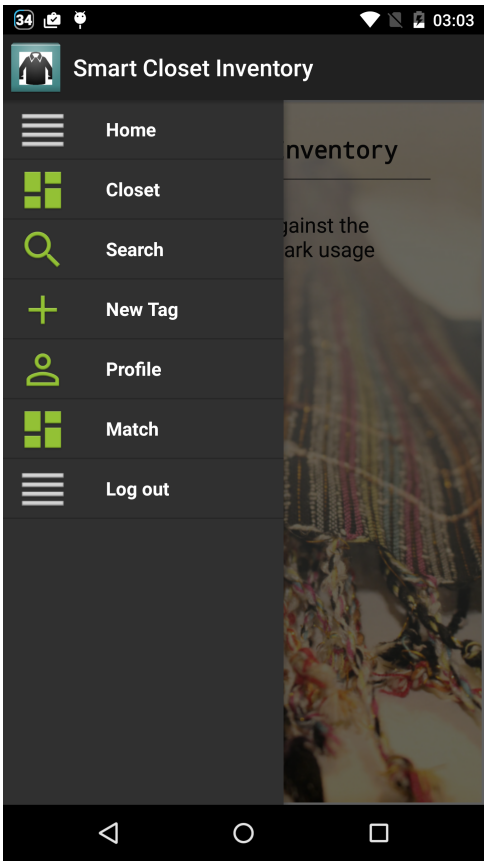

Figure 18: Slider Menu

Figure 19: Home Screen

The slider menu is shown in Figure 18. It provides access to all other screens in the SCI app. Most of the flow within the app is controlled from the slider menu. It provides quick access to main features.

Sign in View

When the user is not logged in, SCI app loads to Sign in fragment. This view gives the user either an option to sign up with Google account or log in as shown in figure 20 and 21.

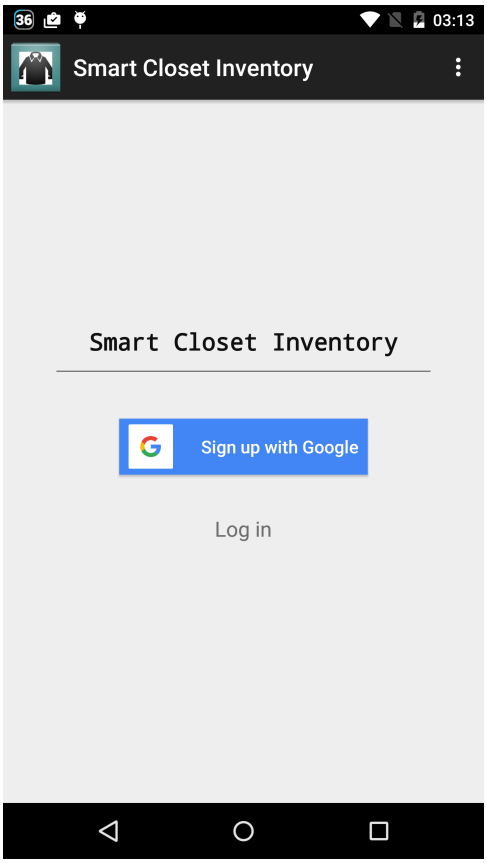

Figure 20: Sign up screen

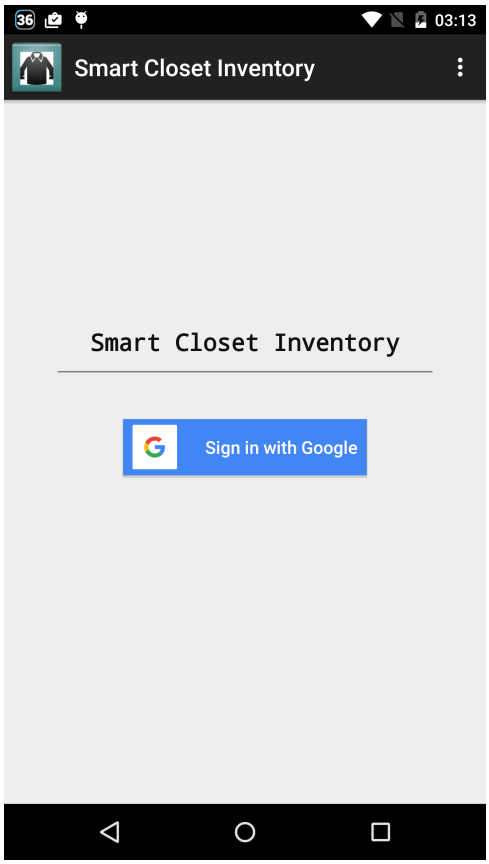

Figure 21: Sign in screen

If the user selects to sign up with Google, the account selection pop up appears. Once the user selects an account, user is logged in. Sign in fragment retrieves user profile information, name, and email from Google profile, and saves it in app shared app preferences. It also calls Create profile backend service to create a user account on the server. The app is redirected to the Home fragment.

If the user selects to log in, Sign in fragment enables the *"Sign in with Google"* button. User is again prompted to pick an account but since this is not new account sign up, no new account is created at the backend. Instead user login is authenticated and app is redirected to the Home fragment.

Closet View

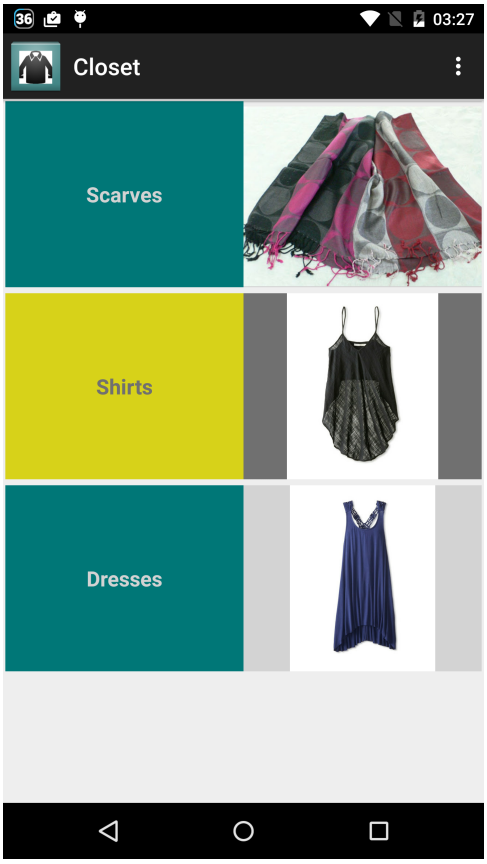

Figure 22: Closet with categories

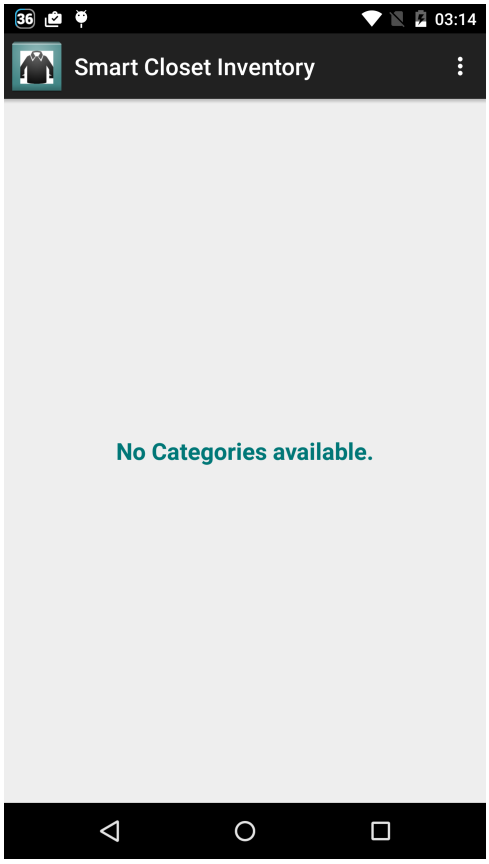

Figure 23: Closet without categories

When the user selects Closet from the slider menu, main activity loads the Closet fragment, which lists all the available categories with an image of most used items. If there are no categories, Closet fragment is loaded with a message as shown in figure 23. It displays the image and category name using a custom List Adapter in scrollable Grid View. This allows listing all available categories for a given user in one fragment.

Category View

When the user selects a category from the closet view, main activity is called notified. Main activity implements fragment interaction listeners and loads the category fragment with items from a given category. This is achieved in similar fashion as closet fragment. Category fragment first invokes a server call to retrieve all articles for a given category. Then, it displays all articles in scrollable custom Grid Adapter.

Figure 24: Category screen

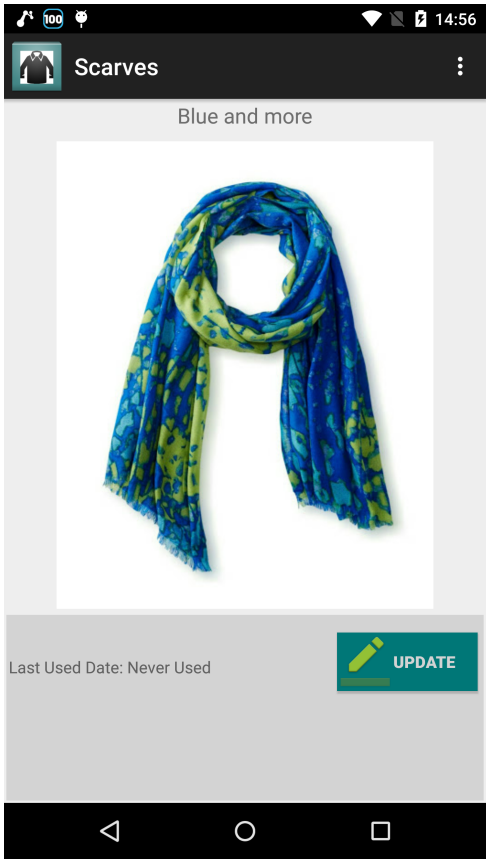

Figure 25: Article screen

Article View

Article view can be loaded from two screens, either tag filter search or from category view. Both search and category fragments call back the main activity to load article fragment with a given article. Article fragment simply loads available article information in the view. Article image is available in URL format. To retrieve the actual image, article fragment utilizes Android `AsyncTask`.

Android `AsyncTask` allows an asynchronous task to run in the background, and its results are published on a UI thread without having to manipulate threads and/or handlers. It is ideal for short operations that last only a few seconds [54]. Since the article image is already available to the article fragment, `AsyncTask` is suitable for quick operations of downloading the article image.

In addition to article image, article fragment also displays the date when the article was last used and an option to update the article.

Update Article View

Update article fragment lists all current article information in an editable view except for the article name that does not allow updates. When the user is done editing, the Save button is clicked to update the article. At this point, update article fragment invokes the backend server to update article information.

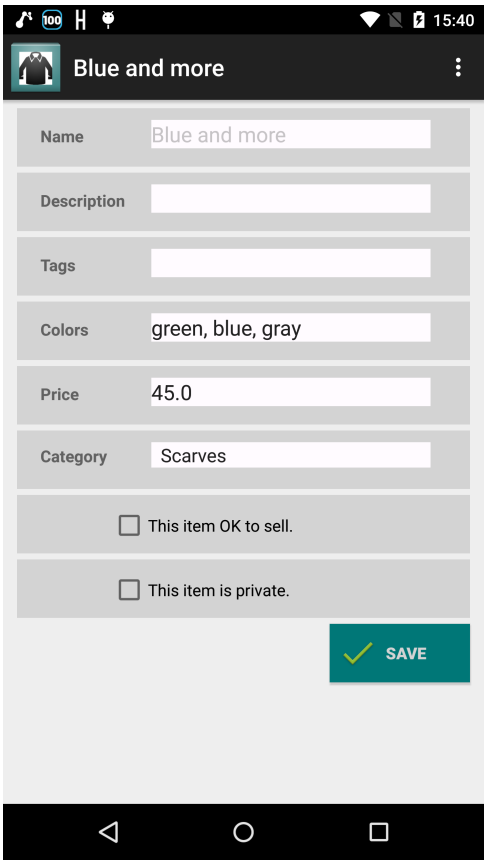

Figure 26: Update Article screen

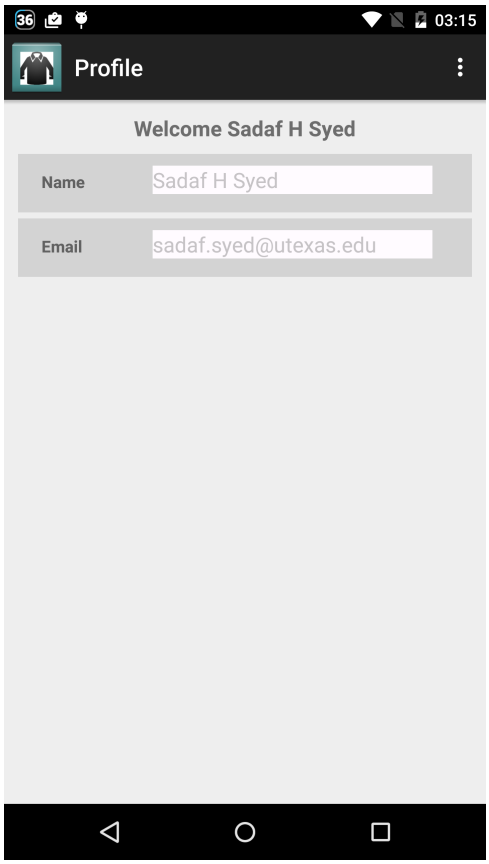

Figure 27: Profile screen

Profile Fragment

Profile fragment displays the user name and email of currently logged in user. This information is stored and retrieved from Android Shared Preferences. Android Shared Preferences points to a simple file on the device containing a collection of key-value pairs. It also provides methods to read and write the key-value pairs [55].

Search View

SCI app supports following five types of search filters:

- Base Search - search based on matching string values with names, tags, and categories
- Usage Filter - filter items are: not used in last 30 days, 6 months, last year, last 2 years, last 3 years, last 4 years or last 5 years
- Tag Filter - finds an article by tag detection
- Never Used Filter - finds all the items that have never been used before, or have been used at least once before
- Selling Filter - finds all the items that are available for sale or not available for automatic sale.

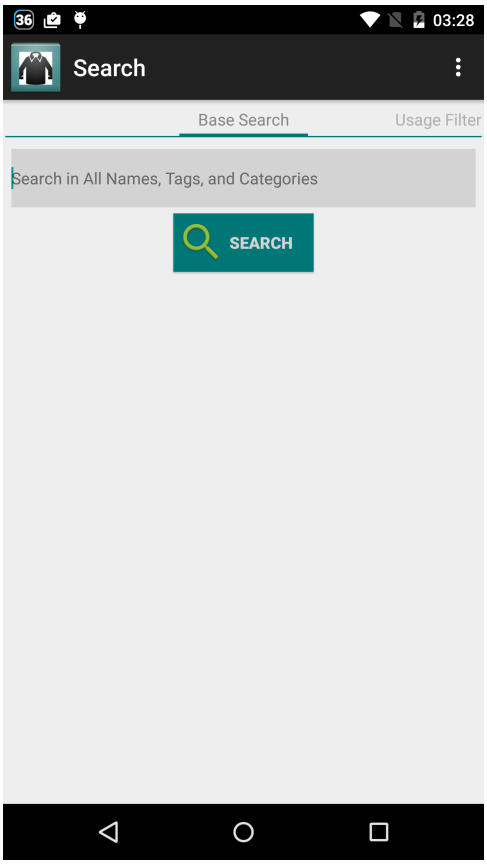

Figure 28: Base Search

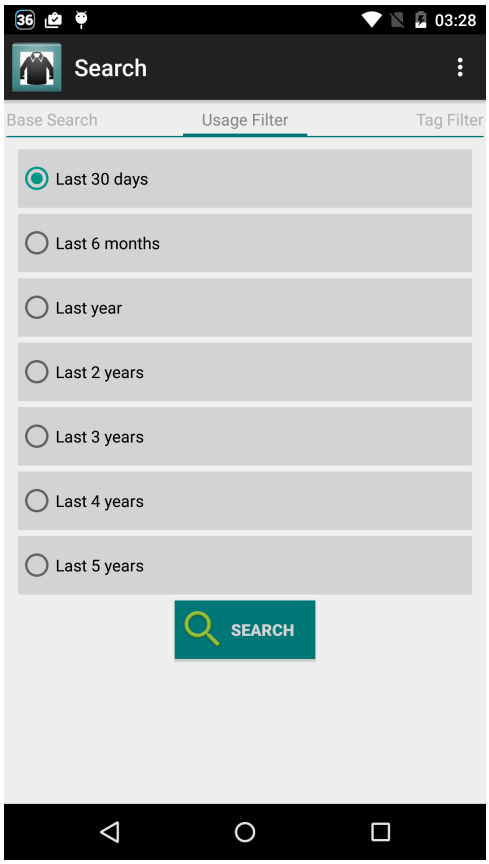

Figure 29: Usage Filter

Search screens are all implemented as View Pager inside the main search tag fragment. Android View Pager gives quick access to user by flipping left and right between the screens. An implementation of Fragment Page Adapter is supplied to generate the all search screen show by the search view [56].

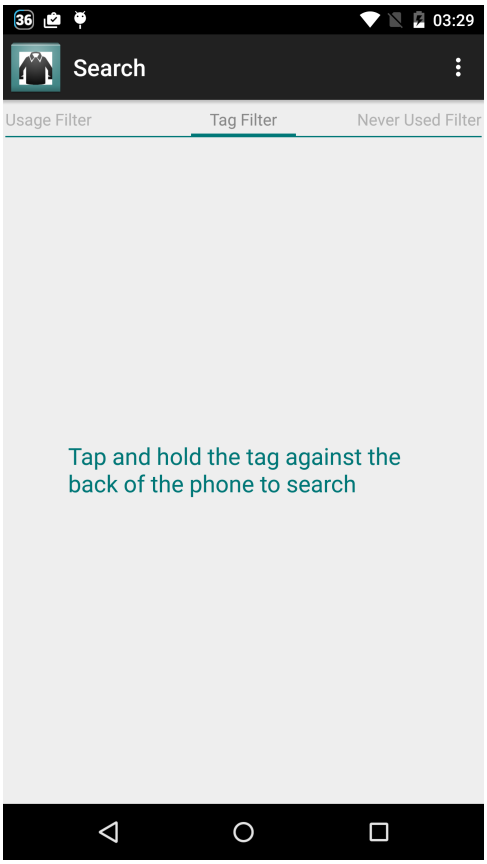

Figure 30: Tag Filter

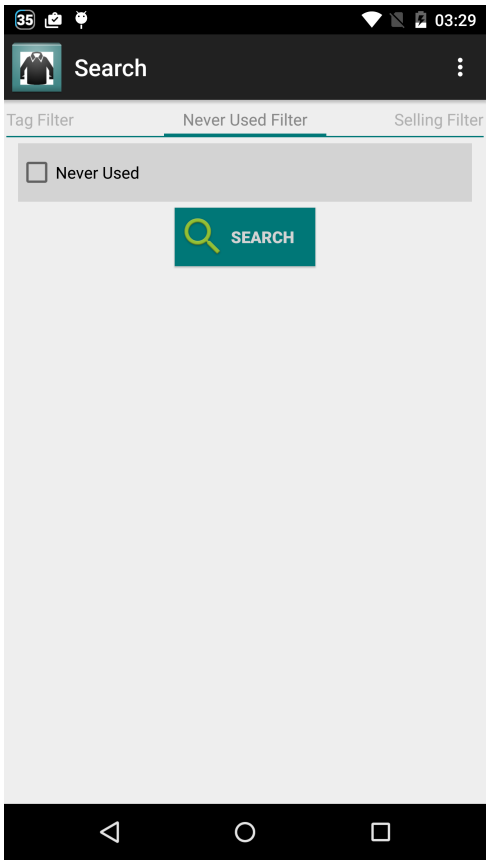

Figure 31: Never Used Filter

Each of the search screens is itself implemented as an Android fragment and is attached to the main activity. When a search is invoked, current search fragment callbacks the main activity to handle the search query. Main activity overrides the fragment interaction listener for each of the search fragments and launches the search with an appropriate search query.

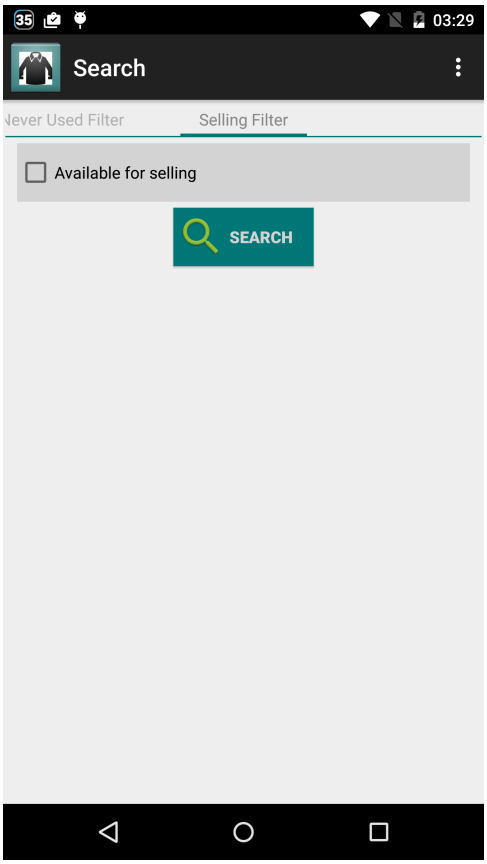

Figure 32: Selling Filter

Figure 33: Search Filter

Search fragment implements a Request Receiver to get back the search results from the server and updates the search view with Grid View listing the articles.

New Tag

The process of creating a new article is divided into three screens:

1. Collecting article information
2. Uploading article image and extracting three dominant colors
3. Writing tag attached to the physical article

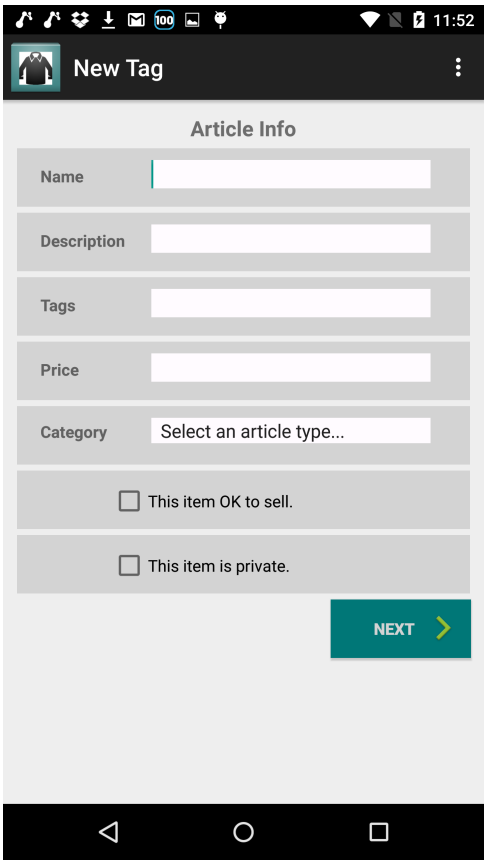

Figure 34: New Tag Article Info

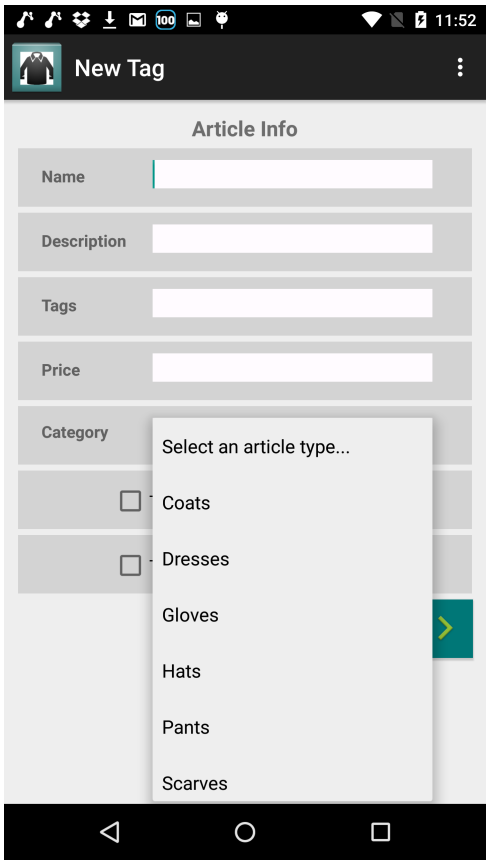

Figure 35: New Tag Categories

Each of the three screens is implemented as Android fragment attached to main activity. Once an article is created, the returned UUID is used to update the article with an image. The Upload Image fragment calls the server running on App Engine to upload an article image and return the newly generated URL. The image URL is then sent to the image processing service running on Amazon Web Services to remove the background and extract three dominant colors. Once the color hex codes are returned by the Amazon Web Services, Upload image fragment invokes update image color service on App Engine to update the articles with extracted colors. The Upload Image fragment performs

all the above server calls in the background, and loads the write tag fragment to write the tag.

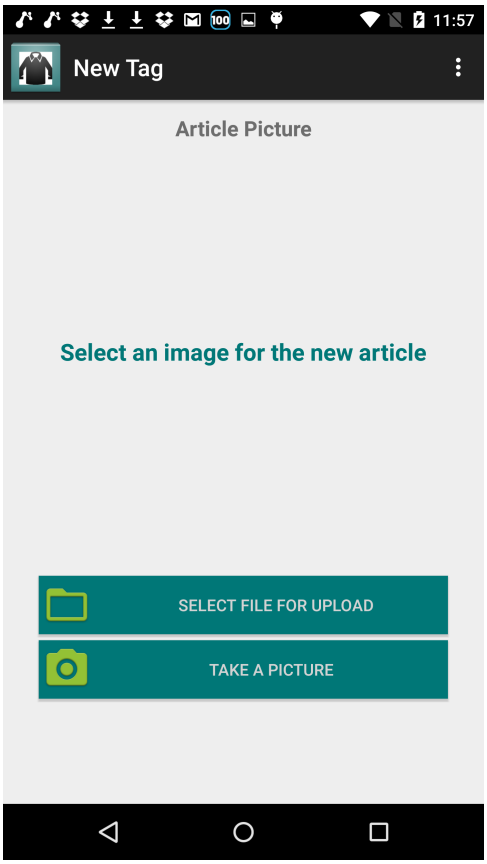

Figure 36: New Tag Article Picture

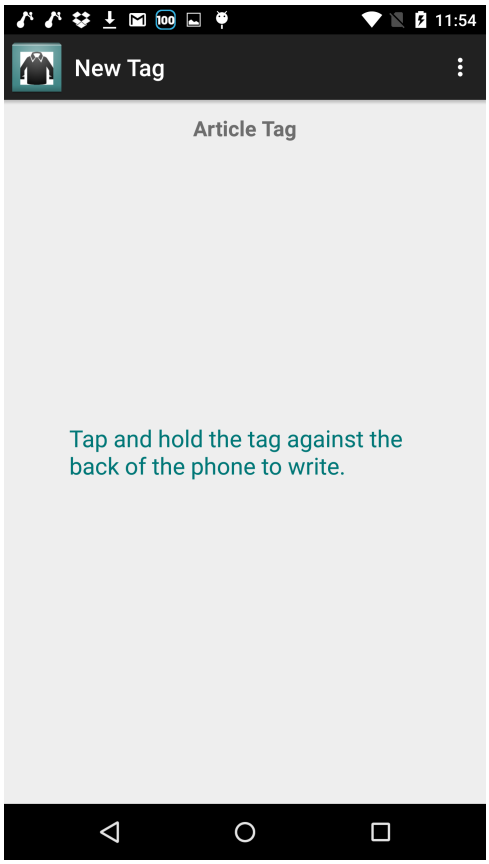

Figure 37: New Tag Scan

Write tag fragment prompts the user to tap the tag against the back of the phone. When the user does so, the article UUID is written on the tag. The actual writing happens in the main activity.

Match View

Match view allows the user to find matching articles from a selected category given an article tag. Match view is divided into three screens:

1. Tag screen - This screen prompts the user to tap a tag for which the user wants to find matching articles
2. Category selection - This screen requires the user to narrow down a category they want to match against
3. Match results - This screen lists the matched articles

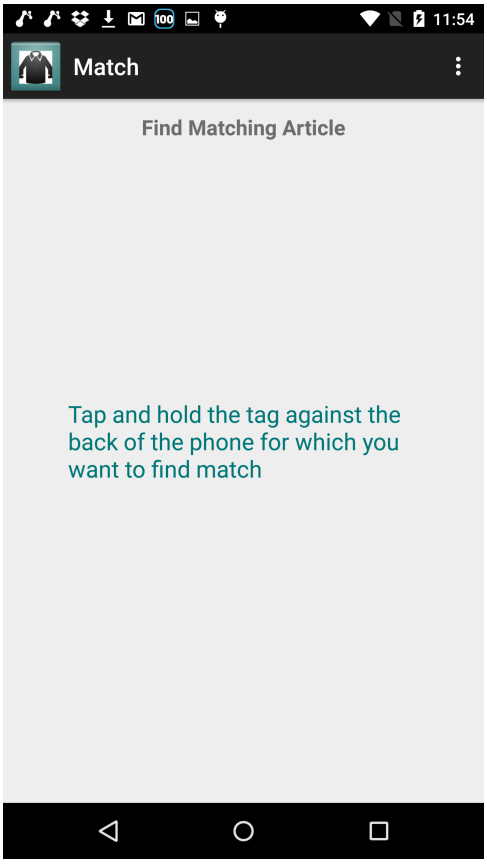

Figure 38: Match tag scan

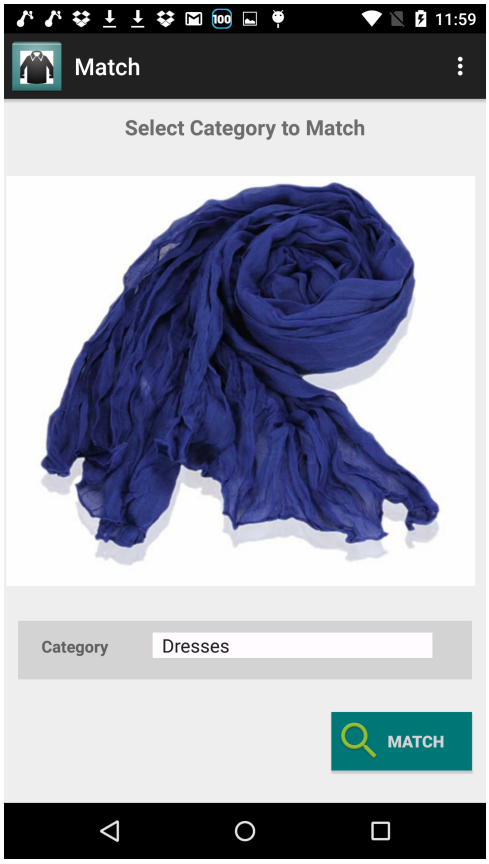

Figure 39: Match category selection

Match view is implemented as a fragment and is attached to the main activity. Upon creation, match fragments load prompts the user to tag the NFC tag attached to an article. NFC tag detection service resides in the main activity. If match mode is enabled, main activity invokes the backend server to retrieve the information about the article tagged. When the article is returned, main activity loads the match fragment again but this time with category selection view. User selects the category in which he wants to find matching articles e.g., find all matching scarves, dresses or hats.

When user clicks Match button, match fragment callback the main activity to launch find match fragment. Find match fragment invokes a backend server call to find the matching articles in a selected category based on the colors of the selected article. The results from the server are displayed in a scrollable Grid View using a custom Grid Adapter to display the name and image of the matched articles.

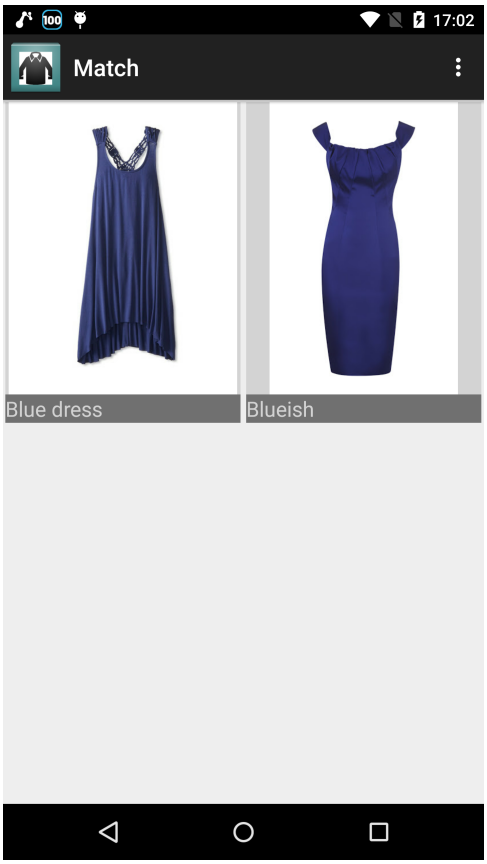

Figure 40: Match results

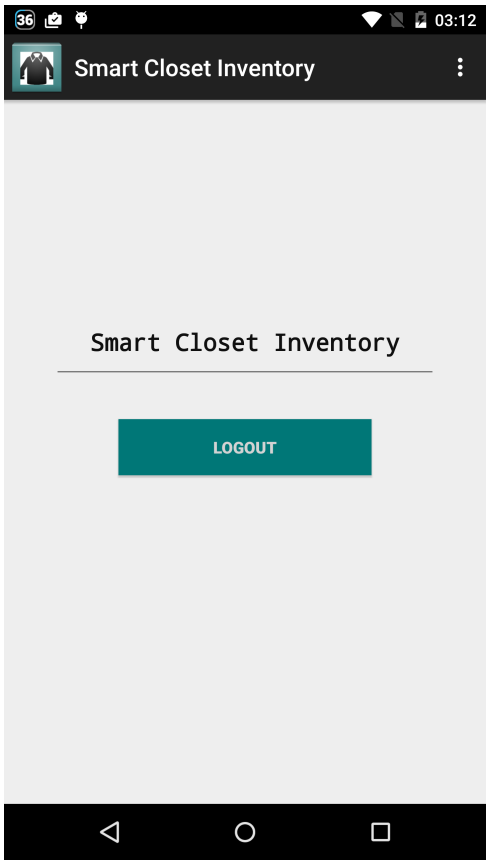

Figure 41: Logout screen

Logout View

Logout view simply gives the user the option to logout of the SCI app. The logging out feature is handled in sign in fragment. When the user confirms the logout action, sign in fragment disconnects the user account using Google Sign-In service. Username, email, and ID tokens are removed from the Shared Preferences. SCI app is then redirected to sign up screen.

Chapter 5: Implementation Notes

This section discusses key aspects of development process including code size, development environments, and common implementation issues.

SECTION 5.1: DEVELOPMENT ENVIRONMENT

SCI Android app development was done on Android Studio based on IntelliJ IDEA [57] with Gradle-based build system. Android Studio has build in support with GitHub [58], which was used to host SCI Android app source repository. It also provides support for downloading Android SDK packages. Although, Android Studio provides optimized emulator image with predefined device profiles based on common Android devices but SCI app testing was done on following Android hardware:

- Nexus 5 running Kitkat 4.4.4 (API level 19)
- Nexus 6 running Lollipop 5.1 (API level 22)
- Nexus 9 running Lollipop 5.1 (API level 22)

SCI backend server and web store, which is hosted on Google App Engine, was mostly developed in Python. For Python editing we used Sublime Text editor [59]. SCI backend server and web store was regularly testing by both local and App Engine deployments using GoogleAppEngineLauncher with Python SDK for Mac OS X [60]. When running locally all the logging is available in GoogleAppEngineLauncher.

SCI image process server deployed on Amazon Web Services is also a Python app and Sublime Text [59] was used for development. Amazon Web Services provides local and Elastic Beanstalk development. Most of the development and testing was done locally using HTTP POST requests. When integrating with SCI Android app, image processing server was deployed on AWS Elastic Beanstalk using AWS and EB CLI [61].

SECTION 5.2: SOFTWARE METRICS

Software Metrics for SCI were computed using CLOC, Count Lines of Code [62]. CLOC is a command line program that counts lines of source code, comment lines, and blank lines in many programming languages [62]. It takes from a single to archive names as inputs for computing lines of code. Following section lists two types of software metrics, one using local git repository and one by individual classes and packages.

Android App Metrics

Figure 42 lists the result of running CLOC on SCI Android app git repo. Java files stat includes all SCI Android app Java code from three packages, `ssar.smartcloset`, `ssar.smartcloset.types`, and `ssar.smartcloset.util`.

```
Sadafs-MacBook-Pro:SmartCloset ssyed$ cloc $(git ls-files)
     103 text files.
     102 unique files.
      71 files ignored.

http://cloc.sourceforge.net v 1.62  T=0.20 s (457.0 files/s, 48773.8 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                            37           1226           1701           4762
XML                             52            253             31           1575
Bourne Again Shell               1             20             21            123
DOS Batch                        1             24              2             64
IDL                              1              2              0             15
-------------------------------------------------------------------------------
SUM:                            92           1525           1755           6539
-------------------------------------------------------------------------------
```

Figure 42: SCI Android app Metrics

CLOC results for all the classes from these packages are sorted in figure 43.

File	blank	comment	code
ssar/smartcloset/MainActivity.java	237	451	847
ssar/smartcloset/SigninFragment.java	105	116	422
ssar/smartcloset/UploadImageFragment.java	88	242	247
ssar/smartcloset/UpdateArticleFragment.java	47	43	201
ssar/smartcloset/NewTagFragment.java	31	32	184
ssar/smartcloset/ProfileFragment.java	36	38	182
ssar/smartcloset/SmartClosetIntentService.java	20	13	182
ssar/smartcloset/UsageFilterFragment.java	42	47	156
ssar/smartcloset/LoginFragment.java	36	38	148
ssar/smartcloset/FindMatchFragment.java	35	53	146
ssar/smartcloset/CategoryFragment.java	37	62	144
ssar/smartcloset/ClosetFragment.java	43	61	143
ssar/smartcloset/SearchFragment.java	35	46	142
ssar/smartcloset/MatchFragment.java	29	31	138
ssar/smartcloset/FragmentRouter.java	31	43	112
ssar/smartcloset/ArticleFragment.java	28	31	105
ssar/smartcloset/NeverUsedFragment.java	22	41	79
ssar/smartcloset/SellFilterFragment.java	23	41	78
ssar/smartcloset/WriteTagFragment.java	17	35	71
ssar/smartcloset/SearchTabFragment.java	20	57	65
ssar/smartcloset/TagSearchFragment.java	15	34	63
ssar/smartcloset/SmartClosetFileService.java	9	11	59
ssar/smartcloset/BaseSearchFragment.java	17	28	55
ssar/smartcloset/CustomOnItemSelectedListener.java	4	7	12
ssar/smartcloset/types/MainMenu.java	8	40	8
ssar/smartcloset/types/CustomListItem.java	1	3	5
ssar/smartcloset/types/Article.java	49	3	163
ssar/smartcloset/types/CustomListAdapter.java	24	12	99
ssar/smartcloset/types/CustomGridAdapter.java	24	10	88
ssar/smartcloset/types/User.java	20	3	70
ssar/smartcloset/types/NavDrawerListAdapter.java	15	3	56
ssar/smartcloset/types/ViewPagerAdapter.java	12	5	51
ssar/smartcloset/types/Category.java	11	3	32
ssar/smartcloset/types/NavDrawerItem.java	5	3	14
ssar/smartcloset/util/JsonParserUtil.java	37	4	140
ssar/smartcloset/util/SmartClosetConstants.java	9	8	44
ssar/smartcloset/util/ToastMessage.java	4	3	11
SUM:	1226	1701	4762

Figure 43: CLOC computation on SCI Android Java packages

Figure 44 shows CLOC computation result on SCI Android resource packages. These include XML files for all the activities, fragments, layout, color, style, and drawables.

File	blank	comment	code
res/layout/fragment_update_article.xml	32	0	128
res/layout/fragment_newtag.xml	30	0	128
res/layout/fragment_create_profile.xml	27	0	107
res/layout/fragment_signin.xml	7	0	86
res/layout/fragment_router.xml	13	0	85
res/layout/fragment_usage_filter.xml	9	0	79
res/layout/fragment_upload_image.xml	8	0	71
res/layout/fragment_article.xml	8	0	63
res/layout/fragment_login.xml	11	0	55
res/layout/fragment_match_filter.xml	8	0	55
res/layout/fragment_profile.xml	10	0	42
res/layout/fragment_closet_item_view.xml	6	0	39
res/layout/fragment_write_tag.xml	5	0	33
res/layout/fragment_base_search.xml	3	0	32
res/layout/fragment_never_used.xml	3	0	30
res/layout/fragment_sell_filter.xml	3	0	30
res/layout/fragment_closet.xml	2	0	25
res/layout/fragment_category_item_view.xml	4	0	24
res/layout/fragment_match.xml	3	0	23
res/layout/fragment_tag_search.xml	2	0	16
res/layout/fragment_category.xml	0	0	9
res/layout/fragment_search.xml	2	1	9
res/layout/fragment_find_match.xml	2	1	6
res/layout/fragment_search_tab.xml	1	0	6
res/layout/activity_main.xml	6	12	27
res/layout/drawer_list_item.xml	3	0	24
res/layout/viewpager_main.xml	4	0	16
res/values/strings.xml	7	1	59
res/values/layout.xml	2	0	30
res/values-v21/styles.xml	2	0	11
res/values/color.xml	0	0	11
res/values/styles.xml	4	1	10
res/values/dimens.xml	0	1	4
res/values-w820dp/dimens.xml	0	3	3
res/values-sw600dp/refs.xml	3	6	3
res/values/refs.xml	1	0	2
res/drawable-hdpi/radiogroup_item_divider.xml	0	0	9
res/drawable/list_item_bg_normal.xml	0	0	7
res/drawable/list_selector.xml	2	0	6
res/drawable/list_item_bg_pressed.xml	0	0	5
res/menu/main.xml	0	0	8
res/menu/menu_write_tag.xml	0	0	6
res/xml/nfc_tech_filter.xml	0	1	6
SUM:	233	27	1428

Figure 44: CLOC computation on SCI Android Resource package

Google App Engine Metrics

Figure 45 shows the results of running CLOC on SCI Google App Engine repository, hosting backend server and web store. Language results for Python includes the original backend server code, 1999 lines of code, test files and all the third party libraries deployment. Results from some of the major libraries are included in figures to follow.

```
Sadafs-MacBook-Pro:storefrontssar ssyed$ cloc $(git ls-files)
     438 text files.
     437 unique files.
      34 files ignored.

2 errors:
Unable to read:  js/jquery-1.11.1.min
Unable to read:  .js

http://cloc.sourceforge.net v 1.62  T=1.70 s (247.0 files/s, 66040.9 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                         364          12882          27586          54274
CSS                             12             28             97           6819
Javascript                      21            989            979           5977
PHP                              2             67            107           1184
HTML                            17            183            104            851
Go                               1             19             10            267
YAML                             3             20              0             75
JSON                             1              0              0             41
-------------------------------------------------------------------------------
SUM:                           421          14188          28883          69488
-------------------------------------------------------------------------------
```

Figure 45: SCI App Engine Metrics

Figure 46 shows CLOC computation on colormath [38] library used by SCI.

File	blank	comment	code
colormath/spectral_constants.py	17	6	845
colormath/density_standards.py	17	9	782
colormath/color_appearance_models.py	226	390	662
colormath/color_conversions.py	208	250	535
colormath/color_objects.py	154	321	352
colormath/color_diff_matrix.py	50	36	85
colormath/chromatic_adaptation.py	28	38	54
colormath/color_constants.py	5	10	39
colormath/color_diff.py	23	42	35
colormath/density.py	13	27	25
colormath/color_exceptions.py	12	15	14
colormath/__init__.py	0	0	1
SUM:	753	1144	3429

Figure 46: CLOC computation on colormath library

Figure 47 and 48 shows CLOC results for GAEUnit [63] and webtest [64] libraries used for SCI integration tests.

File	blank	comment	code
gaeunit.py	78	166	226

Figure 47: CLOC computation on gaeunit library

File	blank	comment	code
webtest/__init__.py	168	310	833
webtest/debugapp.py	3	4	32
SUM:	171	314	865

Figure 48: CLOC computation on webtest library

AWS Elastic Beanstalk Metrics

Figure 49 shows the result of CLOC computation on image processing service deployed on Django server in an AWS Elastic Beanstalk.

File	blank	comment	code
django_eb/django_eb/view.py	45	25	132
django_eb/django_eb/settings.py	29	20	51
django_eb/django_eb/urls.py	3	14	9
django_eb/django_eb/wsgi.py	7	6	6
django_eb/manage.py	3	1	6
SUM:	87	66	204

Figure 49: CLOC computation on SCI image processing service

Development Timeline

This section captures the development timeline for implementing the Android app and backend services. Time spent for each feature is estimated based on the commit history from Git repositories. Table 1 lists time spent for each Android feature. This includes the time spent developing the user interface and integration with the backend service.

Java Classes	Time spent (days)
Android NFC Service	4
NFC write tag	2
Create Article View	5
Closet View	4
Article View	1
Profile View	1
New Tag View	3
Search View	3
Update Article View	4
Login screen	4
Category View	2
Find Match View	5
Google Sign-In Service	7
Authentication	6
UI improvements and icons	2
Total	**7.6 weeks**

Table 1: Development estimates for SCI Android App

Table 2 lists the estimates development effort for each of the backend service.

Services	Time spent (days)
Token Sign in Handler	1
Create Profile Handler	2
Create Article and Upload Image Handler	3
Update Article Image Colors Handler	3
Update Article Handler	3
Read Article Handler	1
Use Article Handler	1
Search Article Handler	3
Get Categories Handler	2
Get Category Handler	3
Find Match Handler	2
Authentication	5
Integration testing	2
Total	**4.4 weeks**

Table 2: Development estimates for SCI backend services

Chapter 6: Testing

App Engine provides testing utilities that use local implementations of its service to enable unit testing and integration testing in a local development environment. SCI heavily depends on the integration testing for its backend server request handlers. These request handlers are a critical integration point in the SCI architecture. Request handlers receive request data from the client, process it, and then return the results. SCI request handlers can easily be tested since they are simple Python classes. The WSGI application wraps these classes in a shell and routes requests to the appropriate handlers. The App Engine provides a similar shell in integration tests [65].

SCI integration tests are implemented using the App Engine testbed service, WebTest and unittest, and the Python unit testing framework. Each of these is discussed below.

SECTION 6.1: APPROACH

App Engine provides service stubs that simulate the behavior of the actual service. For example, the datastore service stub, `init_datastore_v3_stub`, and memcache service stub, `init_memcache_stub`, allows testing the datastore code without invoking any calls to the actual datastore. The entity being tested is held in memory rather than the actual datastore and is deleted after execution of the test [66]. These service stubs are available for unit testing by an App Engine python module called testbed [66].

WebTest

WebTest enables integration testing for WSGI-based web applications by providing a simple interface for executing WSGI applications and verifying the output. Tests executed by WebTest runs validates the full stack of your application by calling the application as a WSGI HTTP server would do without a running HTTP server [64]. This is enabled by TestApp, which is used to wrap the WSGI application to be tested. Once the WSGI application has been wrapped, an HTTP GET request can be issued using `app.get('/')`.

Python unittest

unittest is Python unit testing framework, providing a rich set of classes for creating and executing tests [67]. A test class is created by sub classing `unittest.TestCase`. Each method in this class whose name starts with 'test' is an individual test. `setUp()` and `tearDown()` methods executes setup and cleanup steps before and after the execution of each test method. The results for test case can be verified by using `assertEqual`, `assertTrue`, `assertFalse`, and more. unittest also supports `assertRaises()` to verify specific exceptions are raised successfully [67].

SECTION 6.2: SMART CLOSET INVENTORY - CREATE ARTICLE TEST EXAMPLE

This section covers an example of the create article request handler test which deals with both the handler and the datastore service.

Figure 50 shows test case initializes the SCI WSGI application that uses the create article request handler and wraps this app in WebTest. Next, a test case creates an

instance of Testbed and activates it. Testbed is deactivated in the `tearDown()` method. Test case also initializes the relevant service stub, `init_memcahce_stub`.

```
def setUp(self):
    app = webapp2.WSGIApplication([(r'/CreateArticle', storefrontssar.CreateArticle)])
    self.testapp = webtest.TestApp(app)

    self.testbed = testbed.Testbed()
    self.testbed.activate()

    self.testbed.init_memcache_stub()

def tearDown(self):
    self.testbed.deactivate()
```

Figure 50: Create Article test case set up and tear down

In test method, an article JSON is created, and a POST request is made with simple `post()` call. The return JSON from create article service returns either the UUID of newly created article or an error code as the `returnval` parameter.

To verify a POST result, `returnval` is parsed from the response JSON and asserted for failure codes. If `returnval` is equal to any of the defined error codes, the test case fails. The create article request handler validates the ID token before creating the article. This ID token validation is skipped in integration test cases. Instead Client ID is used as the ID token.

SECTION 6.3: TESTING FRAMEWORK

Across twelve APIs, SCI integration test suite implements 22 test cases. These test cases are executed using GAEUnit [63] App Engine testrunner. GAEUnit allows the test execution in a real GAE app server environment using a web browser. The test execution is also supported in local GAE app deployment as well.

GAEunit framework is quick and easy to setup. The test modules are placed under the 'test' directory along with *gaeunit.py* in the web app root directory. By default, GAEUnit picks up all the modules in the 'test' directory with `TestCase` class. Once the application is deployed locally, test cases can be executed by visiting: *http://localhost:<port>/test*.

DISCUSSION

This section is divided into three parts. It starts with the discussion of future extensions to SCI in Chapter 7, moving on to the lesson learned and conclusion in Chapter 8. In Chapter 9 discusses three current solutions for closet management and how SCI is an improvement over these solutions.

Chapter 7: Future Work

There are many extensions to SCI app and web store that can further improve user closet organization and styling choices. In the section below we discuss some of the exciting new features that extend current functionalities.

SECTION 7.1: TAGS EXTENSIONS

Support for RFID and QR tags

In addition to NFC tags, SCI can be enhanced to support RFID and QR tags as well. The main reason for selecting NFC tags was the ease of scanning. However, QR and RFID tags have their own advantages. While QR codes readers are sensitive to reading conditions, QR codes are cheaper and could be printed on clothing items. RFID tags are also available for clothes [68]. This feature will allow user to use multiple kinds of tags while still allowing use of all the SCI features.

Support to disable tags

Although real time tracking based on NFC tags is the main feature of SCI, it can still be provided as an optional feature. Consider a use case where a user does not have an NFC tag, but would like to be able to create articles in the closet without tags. Also, when the NFC tags feature is disabled, the user should be able to log article usage.

SECTION 7.2: SOCIAL EXTENSIONS

Closet Sharing Support

Currently, closet articles can only be shared on the SCI web store. This sharing is further restricted to everyone or no one. The sharing feature could be added to SCI Android app as well. The SCI app could further allow users to connect with each other and share individual articles, categories, or the entire closet.

SECTION 7.3: IMAGE PROCESSING EXTENSIONS

Outfit tagging

In addition to just individual article inventory, a user could be allowed to tag a complete outfit. SCI will allow the user to upload an image for an outfit. Image processing algorithms could be applied to extract dominant colors from outfit images. This info will be saved along with the creation date of each outfit. Also, the user could share their outfits with other users and get their opinions.

Color trends and outfit suggestions

This feature is a further extension of the outfit tagging feature. As a user starts tagging more and more outfits, SCI will learn color trends they are interested in when selecting an outfit. Later on SCI can give suggestions when a user is creating a new outfit based on the user's color trends. The user will start tagging by outfit and then select categories they want to include. SCI will come up with articles similar to the user's preferred color trends. The user can either accept the changes or decline them. Once the user is satisfied, they can save all articles for the new outfit. Users can manually update

this outfit further and eventually upload an image of the outfit. This will further let SCI learn about user color trends.

Smart Shopping

Humans tend to repeat what they like. We tend to eat the same food we like, play the same games, and even buy the same clothes. SCI can further improve a user's clothing choices. The current color matching feature of SCI can be further enhanced and extended to match new items that do not exist in the user's closet. For example, when a user is planning to buy a new sweater, they can take a picture of the item and match it against their closet inventory. They can dismiss the idea of purchase if a similar item already exists in their closet or purchase it instead if they can find a matching outfit to go with it.

Chapter 8: Conclusion

The rapid growth and advancements in the IoT field is connecting more aspects of human life to the Internet. With a few quick touches, we archive pieces of our lives to the Internet every day. However, certain aspects of our life are inherently physical, such as the people we meet, food we eat, clothes we wear, and places we visit. Every day more IoT devices are integrating these interactions with the Internet so we can store memories, share them with others, and improve our quality of life.

Smart Closet Inventory system allows users to organize their closet by tracking infrequently used articles and improving clothing choices. With the use of NFC technology, SCI allows users to quickly scan their clothes, track what they used, when they used it, and how many times it has been used. SCI helps users utilize all items in their closets and discard unused items. SCI also uses collected data to provide an advanced search based on real-time usage data and color.

While the Smart Closet Inventory system is a great improvement over pre-existing apps, SCI has the potential for endless enhancements. This first iteration of SCI provides basic building block features for managing inventory and gathering data. Leveraging the currently collected data in future extensions to share and track user trends could further improve the way we dress and stay connected, some of the most essential aspects of life.

The development of SCI taught us many lessons. We learned feature-based development is the best approach to develop a feature-rich system such as SCI. Hence, it is crucial to identify the feature dependencies, e.g., the clothing suggestion feature depends on data analysis, which in turn depends on data extraction. In addition to utilizing feature-based development, we also learned that there are a great amount of

existing libraries for image processing in Python and C. Utilizing current image processing libraries not only speeds up the development process, but also increases the robustness of the system, as these libraries have already been researched, developed, and tested.

Finally, existing libraries bring with them a number of constraints. For the SCI application, Amazon Web Services provides a wide range of runtime environments along with data storage options. Selecting an open environment from the beginning can reduce time-intensive research and development.

Chapter 9: Related Works

There are many closet/wardrobe organizing apps available, but no other app supports tagging with NFC tags to track real time usage or matching. Selected products are discussed in this section:

SECTION 8.1: CLOTH APP

The Cloth app is a photo based app that lets the user archive an entire wardrobe as outfits and categorize them by events, weather, location, designer, color, mood, and more. It helps the user figure out what to wear depending on previous outfits they have saved. It also aims to help the user fill in the gaps with new items or make new outfits based on what they already have [69].

In contrast to Cloth, SCI archives all items (including ones that have never been used), not only articles that are part of an outfit. Although outfit tracking is a nice to have feature, it does not provide exact usage of a given article. Also, items need to be captured in a previous outfit in order to add it inventory. Therefore, no unused items can be tracked. The SCI built in search filter can quickly return a list of all items that have never been used.

SECTION 8.2: NETROBE

Netrobe provides many similar features to SCI. It helps users get organized by creating a virtual archive of all items. It organizes items based on category for quick access. It also lets the user remove the background from the image of an item. Moreover, it lets the user create outfits from the existing archive [70].

The only feature lacking in Netrobe is that it does not track usage of the items in the closet. In addition to a digital archive of the closet, SCI lets the user mark the usage of any item by simply scanning the tag. SCI also extracts colors from clothing items automatically when an image is loaded and archives them, minimizing the impact of future queries.

SECTION 8.3: STYLEBOOK

Stylebook is an iOS app for creating a virtual wardrobe and storing it locally on the device. It lets the user add items with images and manually remove the background, which could often be a cumbersome process [71]. It tracks the usage of items and how many times they were worn by tracking if and when an item was added to a calendar.

Unlike SCI, Stylebook requires users to add each item on the calendar to mark the usage. This requires the user to first add the item to an outfit and then add it to the calendar. SCI as previously mentioned speeds up usage tracking with the use of NFC tags. Also, SCI archives the user's closet on the server, therefore eliminating the requirement to backup the device or sync it to different devices.

References

1. Cisco Inc. *Internet of Things (IoT)*. Available from: http://www.cisco.com/web/solutions/trends/iot/overview.html.
2. *Current World Population*. Available from: http://www.worldometers.info/world-population/.
3. *Why fitbit*. Available from: https://www.fitbit.com/whyfitbit.
4. *Vessyl*. Available from: https://www.myvessyl.com/vessyl/.
5. *iluminate*. Available from: http://iluminate.com/.
6. Faulkner, C. *What is NFC? Everything you need to know*. November 17, 2015; Available from: http://www.techradar.com/us/news/phone-and-communications/what-is-nfc-and-why-is-it-in-your-phone-948410/3.
7. Atzori, L., A. Iera, and G. Morabito, *The Internet of Things: A survey*. Computer Networks, 2010. **54**(15): p. 2787-2805.
8. *Contactless mobile payments (finally) gain momentum*. Available from: http://www2.deloitte.com/global/en/pages/technology-media-and-telecommunications/articles/tmt-pred-contactless-mobile-payments.html.
9. Pampattiwar, S., *Literature Survey on NFC, Applications and Controller*. International Journal ofScientific & Engineering Research, 2012. **3**(2).
10. *NFC Forum Technical Specifications*. Available from: http://members.nfc-forum.org/specs/spec_list/.
11. Haselsteiner, E. and K. Breitfuß. *Security in near field communication (NFC)*. in *Workshop on RFID Security RFIDSec*. 2006.
12. Coleman, D., B. Jepson, and T. Igoe, *Beginning NFC*. 2014: O'Reilly Media
13. *Type 2 NFC Tag (Laundry Token) - Circle (30mm)*. Available from: http://nfctags.tagstand.com/products/type-2-nfc-token-laundry-ultralight-c-circle-30mm?variant=3173627905.
14. *NFC Laundry Token Advanced (10 pcs package)*. Available from: https://www.nfc-shop.net/NFC_Laundry_Token_Advanced_p/nst-ur18a.htm.
15. *NFC Laundry Token - NTAG213*. Available from: http://buynfctags.com/nfc-laundry-token-ntag213/.
16. Google Inc. *Python Runtime Environment*. September 9 2015; Available from: https://cloud.google.com/appengine/docs/python/.
17. *Magick++ API for GraphicsMagick*. Available from: http://www.graphicsmagick.org/Magick++/.
18. Google Inc. *The Python NDB Datastore API*. October 29, 2015; Available from: https://cloud.google.com/appengine/docs/python/ndb/
19. *Explaining the webapp2 Framework*. October 29, 2015; Available from: https://cloud.google.com/appengine/docs/python/gettingstartedpython27/usingwebapp.
20. *List of NFC-enabled mobile devices*. Available from: https://en.wikipedia.org/wiki/List_of_NFC-enabled_mobile_devices.

21. *NFC Enabled Phones And Tablets*. Available from: http://rapidnfc.com/nfc_enabled_phones.
22. Goggle Inc. *Near Field Communication*. Available from: http://developer.android.com/guide/topics/connectivity/nfc/index.html.
23. Google Inc. *NFC Basics*. Available from: http://developer.android.com/guide/topics/connectivity/nfc/nfc.html#dispatching.
24. Ellis, C. *Image Background Removal*. 2014; Available from: http://developers.lyst.com/2014/02/13/background-removal/.
25. *Color Survey Results*. Xkcd; Available from: http://blog.xkcd.com/2010/05/03/color-survey-results/.
26. Bell, E. *Color Detection*. Available from: http://developers.lyst.com/2014/02/22/color-detection/.
27. *pgmagick 0.5.1 documentation*. Available from: https://pgmagick.readthedocs.org/en/latest/.
28. *Magick::Image Class*. Available from: http://www.graphicsmagick.org/Magick++/Image.html.
29. *Using python and k-means to find the dominant colors in images*. October 23, 2012; Available from: http://charlesleifer.com/blog/using-python-and-k-means-to-find-the-dominant-colors-in-images/.
30. Luccheseyz, L. and S. Mitray, *Color image segmentation: A state-of-the-art survey*. Proceedings of the Indian National Science Academy (INSA-A), 2001. **67**(2): p. 207-221.
31. Park, S.H., I.D. Yun, and S.U. Lee, *Color image segmentation based on 3-D clustering: morphological approach*. Pattern Recognition, 1998. **31**(8): p. 1061-1076.
32. Weeks, A.R. and G.E. Hague. *Color segmentation in the hsi color space using the k-means algorithm*. in *Electronic Imaging'97*. 1997. International Society for Optics and Photonics.
33. Segaran, T., *Programming Collective Intelligence*. 2007: O'Reilly Media.
34. *The 954 most common RGB monitor colors, as defined by several hundred thousand participants in the xkcd color name survey*.; Available from: http://xkcd.com/color/rgb/.
35. *Lab color space*. October 26 2015; Available from: https://en.wikipedia.org/wiki/Lab_color_space.
36. Rys, R. *What is lab color space?* ; Available from: http://hidefcolor.com/blog/color-management/what-is-lab-color-space/.
37. *gtaylor/python-colormath*. March 20 2014; Available from: https://github.com/gtaylor/python-colormath/blob/master/colormath/color_diff_matrix.py.
38. *Python Color Math Module (colormath)*. Available from: https://github.com/gtaylor/python-colormath.
39. *Django*. Available from: https://www.djangoproject.com.

40. Google Inc. *Authenticate with a backend server*. November 2, 2015; Available from: https://developers.google.com/identity/sign-in/web/backend-auth.
41. *Shop Item*. Available from: http://startbootstrap.com/template-overviews/shop-item/
42. *Bootstrap*. Available from: http://getbootstrap.com/
43. *Bootsnipp*. Available from: http://bootsnipp.com/
44. Google Inc. *Fragments*. Available from: http://developer.android.com/guide/components/fragments.html.
45. Google Inc. *Activity*. Available from: http://developer.android.com/reference/android/app/Activity.html#Fragments
46. Google Inc. *IntentService*. Available from: http://developer.android.com/reference/android/app/IntentService.html.
47. *Android IntentService example using BroadcastReceiver and update Activity UI*. Available from: http://www.mysamplecode.com/2011/10/android-intentservice-example-using.html
48. Singh, I., J. Lietch, and J. Wilson. *Gson User Guide*. Available from: https://sites.google.com/site/gson/gson-user-guide.
49. Steele, J., *The Android Developer's Cookbook: Building Applications with the Android SDK*. 2nd ed. 2013: Addison-Wesley Professional.
50. Google Inc. *Advanced NFC*. Available from: http://developer.android.com/guide/topics/connectivity/nfc/advanced-nfc.html#foreground-dispatch.
51. Google Inc. *Start Integrating Google Sign-In into Your Android App*. Available from: https://developers.google.com/identity/sign-in/android/start-integrating.
52. Google Inc. *Integrating Google Sign-In into Your Android App*. November 11, 2015; Available from: https://developers.google.com/identity/sign-in/android/sign-in.
53. Google Inc. *GoogleApiClient*. November 5, 2015; Available from: https://developers.google.com/android/reference/com/google/android/gms/common/api/GoogleApiClient.
54. Google Inc. *AsyncTask*. Available from: http://developer.android.com/reference/android/os/AsyncTask.html.
55. Google Inc. *Saving Key-Value Sets*. Available from: http://developer.android.com/training/basics/data-storage/shared-preferences.html.
56. Google Inc. *ViewPager*. Available from: http://developer.android.com/reference/android/support/v4/view/ViewPager.html
57. *The Most Intelligent Java IDE*. Available from: https://www.jetbrains.com/idea/.
58. *GitHub*. Available from: https://github.com/.
59. *Sublime Text*. Available from: http://www.sublimetext.com/
60. Google Inc. *googleappengine* Available from: https://code.google.com/p/googleappengine/.

61. *AWS Elastic Beanstalk*. Available from: http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3.html.
62. *CLOC* Available from: http://cloc.sourceforge.net/#Languages
63. *Gaeunit*. Available from: https://code.google.com/p/gaeunit/
64. Bicking, I. *Testing Applications with WebTest*. Available from: http://webtest.pythonpaste.org/en/latest/.
65. Google Inc. *Handler Testing For Python*. September 22, 2014; Available from: https://cloud.google.com/appengine/docs/python/tools/handlertesting.
66. Google Inc. *Local Unit Testing for Python*. October 29, 2015; Available from: https://cloud.google.com/appengine/docs/python/tools/localunittesting#Python_Introducing_the_Python_testing_utilities.
67. *The Python Standard Library*. Available from: https://docs.python.org/2/library/unittest.html.
68. *Invengo Gen2 Monza4QT Apparel Tag with 3D and Privacy Controls*. Available from: http://www.rfidtags.com/invengo-monza4qt-apparel-tag.
69. Shatzman, C., *The New Cloth App Makes Your Virtual Closet Dreams Come True*, in *Forbes*. 2014.
70. *What to Wear: Apps for Managing Your Closet*. February 14, 2013; Available from: http://allthingsd.com/20130214/what-to-wear-apps-for-managing-your-closet/
71. Caitlin. *There's An App For That: Stylecook V. Closet+*. Available from: http://www.andpossiblydinosaurs.com/theres-an-app-for-that-stylebook-v-closet/.